Shards: A System for Systems

A thesis submitted in fulfilment of the requirements for the degree of
Master of Computer Science

Andrew Shelton
B.App.Sc. (Hons)

School of Computer Science and Information Technology
RMIT University

December 2013

Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged. The work has been carried out since the beginning of my candidature.

Acknowledgements

My deepest thanks to Dr. James Harland, without whom this thesis would never have reached maturity.

I'd also like to thank my parents, Mike and Betty, for their support during this thesis's overly extended gestation period. And even greater debt is owed to my beloved wife Dr. Wendy Chew for her tolerance and support during the further extended duration.

Andrew Shelton
School of Computer Science and Information Technology
RMIT University
September 14, 2014

Contents

List of Figures

List of Tables

Abstract

This thesis considers the question of optimising what is generally referred to as systems software and often considered synonymous with operating systems. These systems tend to have many internal subsystems that act semi-independently but cooperate for a common system goal. This goal is generally the management of resources and the provision of services to some other software process. Systems software tends to be large, commercially valuable and exhibit complex behaviours since it must balance available resources against client demands. This allows for a variety of strategies and optimisation possibilities to influence the design of the system. The advantages of the system software can then be leveraged by the software they support.

This approach is distinct from execution optimisation. Code optimisation at the micro-level is well understood. Languages, compilers and libraries are extremely clever in converting human instructions into efficiently executable code. This is partly made possible by the languages being designed to assist the compiler, languages having a long active lifetime and the optimisations being focused on the reasonably controlled environment of program execution. It is assumed this technology will be employed as a foundation.

At a higher design level, the code will have an internal structure. It will have multiple modules, processes or executables that must cooperate. Good design encourages the selection of appropriate data structures and algorithms but there is little assistance in optimising the interactions and sequences within the system. Over time systems have become larger which exponentially increases the number and extent of interactions within the system. This has increased the difficulty in understanding let alone optimising these interactions.

These increasingly complex systems are also expensive to build. The amount of effort required to manually analyse and optimise each internal operation and interaction is likely to substantially increase the time and effort required in construction. The viability of constructing the system will be influenced by this increase in resources needed. In addition, much of the

final form and fine detail of the system may be unclear until construction is completed. In such a situation it is pragmatic to focus on implementation and leave questions of system level optimisation for later.

For many systems, optimisation may be a low priority. Powerful and low cost computing platforms can make a design with many sub-optimal interactions sufficient for the task. If performance is a problem then focused optimisation on revealed issues in the running system will often be sufficient. In addition many specialised systems, when compared to a programming language, have a relatively small audience and short life-time.

The systems of interest for this thesis will be those where the system is constructed for a specific goal. This allows performance optimisation to be measured against these goals. They will have a high commercial value, be widely deployed or have a long operational life such that effort invested in optimisation will have a opportunity to pay dividends.

Operating systems are perhaps the best known field in which this combination of factors can occur. Embedded systems, which may not have a formal operating system, are another potentially large domain to consider. Systems software can also exist as the middleware in large and layered software architectures.

It would be ideal to have a fully automated system optimiser but this would be extremely difficult outside of what is already done by the compiler. These systems are large, complex and most importantly contain a great capacity for design trade-offs. An optimisation only has value in terms of how its strengths and weakness interact with the structure of the system and the design goal for which it is being constructed. In addition being a designed system there is the possibility of restructuring the system to expose or allow new optimisation possibilities.

This thesis explores a new approach for optimisation at the level of code modules. This approach has the potential to make system construction faster and thus cheaper, increasing the range of tasks for which system construction may be economically viable. It needs to do this in comparison with the current model of hand-crafted bespoke system programming. The approach also needs to avoid restricting performance efficiency, solution flexibility or increasing complexity through a profusion of special cases and specific variants. This is not generally true of current attempts to provide a universal set of modules for system construction. This is partly because there is an inherent conflict between reusability, simplicity and optimisation to a specific environment.

The approach taken in this thesis is to consider that a module is primarily conceptual. It should not have a static definition or rely on a run-time determination of how it will operate as these impact flexibility and performance. Instead, it will be information on how to automatically select one of a family of implementations for the functionality it represents based on what other components exist within the system. In addition, it will tailor itself based on the components it interacts with and information provided by the creator on the design goals and intended tasks for which the system will be used. This is then used to construct a system from these constructed components.

An example would be a system that is going to exist within a remote probe. There are a great number of things it will not need, such as a local user interface, and each component should respond to that absence in terms of which implementation is selected, how it is tailored and how it will be integrated. The designer may decide it does not need multitasking, that it will only ever run one program (the "be a probe" program), that program is known in advance and also that some hand-crafted pieces are provided for few very specialised and system specific needs. A smart component could respond to each of these possibilities in terms of optimising both itself and the system of which it will be a part.

A process for generating optimised operating systems is an extremely large scale goal in terms of the range of possible systems and the environments in which they will operate. This is another reason to favour a modular approach such that the process can be evolved iteratively and value given to the system by reuse of general mechanisms and specific modules in later work. It also means the systems need not be complete. Where no module exists code will be constructed as normal and tied into the system, possibly creating a new module in the process. This approach also has the potential to make experimentation and study in the field of systems more productive should it reach maturity.

Chapter 1

Introduction

Operating systems, and systems software in general, are a fundamental foundation of the modern computing environment. Their original purpose was a result of having to join the products of computer hardware and software engineering into a harmonious system that enabled the best results for both parts. As such they provided a design challenge in terms of hardware resources, the demands of software clients and the over all goal of the system being constructed. A well built system provided a coherent model, a clean interface and efficient mechanisms in a unified package.

The original operating systems focused primarily on making the functionality of the hardware more accessible to users. The combination of hardware, operating system and probably some system applications made up a *computing system* and could be sold more broadly and used more productively than the hardware alone. As computers became more powerful and software sophistication increased the exact definition of *systems software* became more complicated. In this thesis the term systems software will be used in the broadest sense of software that manages a layer between a resource and a collection of clients.

> "How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Computer hardware is constructed towards this goal. Since bare hardware alone is not particularly easy to use, application pro-

> grams are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system."
>
> - *Silberschatz et. al.* [Silberschatz et al., 2013]

Operating systems tended to be complex software products and once mature were expected to recover the costs of their construction over a long period of use. There was rarely strong commercial incentive to radically improve or alter them. Creating a new system from scratch would be a substantial amount of work and unlikely to be competitive with the mature systems already in the market.

Operating systems would still evolve over time. New hardware could force changes or users could find faults needing repair. A system designer might even examine the system to see if subsections of code could be improved to increase efficiency. These changes would face a significant disincentive if they were significant or changed the behaviour of the system. A change could introduce new faults and negative side-effects that could place the stability of working systems at risk. In addition existing programs, developers and even system maintainers rely on the stability of system behaviours and interfaces for their work. A change that could be avoided, worked around or performed without disturbing the system model and interfaces had many advantages.

From the system designer point of view there is a tension between the creation of a new system and the evolution of an existing system. A truly novel system provides the greatest opportunity for innovation but is rarely practical as a commercial or research project. An element within an existing system is more manageable and easier to get into the field but is also much more constrained. Outside of some truly novel algorithm there is less chance of making an significant contribution in place of an already functioning sub-component.

This is also reflected in how much support exists for the process of system construction. When working within an existing system the framework is fully specified and relatively inflexible. For a new system there is no support at all. The design process starts with some general models of how an operating system works (often derived from existing systems) and a blank piece of paper or empty source code file. It seemed desirable to consider if the process of system design could

itself be the subject of investigation.

The challenge was to consider systems as being structures built from parts which could be considered in their own right. The existing approach of inventing and implementing a novel fundamental algorithm could be considered as improving a part. The improvement would be measured in terms of superior results in the benchmark for that operation. This investigation would focus on the system as a whole. Could parts of existing systems be reused, could new structures of pieces be easily integrated and would the system as a whole exhibit new capabilities to justify the change.

The term *Shard* was selected in advance for this idea of a reusable piece of a larger system. The focus was a process in which new systems could be expressed: a system of systems.

1.1 An Introductory Scenario

Consider a large software project; It has various resources, which in software is primarily reflected in the dedicated time of skilled professionals. The project group has been set a complex task and has been given an amount of time in which it is expected that a solution to the problem can be completed. The first steps are going to involve planning so that time, both in terms of people and the project as a whole, are used productively. All elements of the project will be considered because the value of the solution depends on it being a complete solution.

At some point in time a decision will be made on whether the system software will be relevant to the project. This includes all software used to support the development and construction of a solution such as compilers, libraries, operating systems and middleware. The importance of this area will be heavily dependent on the nature of the project. The majority of projects are not dependent on the specific properties of the software infrastructure required to support the execution of their application. These system software components must exist but the relative costs and benefits of the different possibilities are sufficiently minimal or unclear that any selection is acceptable. For such projects system software which is known to work, well understood and widely deployed is a sufficient basis for selection.

However some subset of projects will exhibit deeper interaction with systems software. Their success will be made easier or harder by the properties of the underlying infrastructure. An example of this is software for mobile devices which can find themselves forced to consider remote computing, peer-to-peer resource sharing, limited processor power, restricted interfaces

and the fact that every facility exacts a cost in battery power. These projects may also seek competitive advantage in novel solutions, which likely involves both a deeper dependence on the system and a reduced time in which to carry out the project.

Ideally, these needs will be satisfied by finding an existing operating system that supports the necessary qualities. There are a variety of systems that have evolved to support specialised domains. These are still general in that they are not specific to the exact needs of the project. Instead, these systems are customised to the needs expected to be common to a product domain. These systems will be designed to focus on capabilities that are needed and will limit or omit capabilities that are not. The interesting cases occur at the boundaries where neither the general nor the domain solution are ideal. It may also occur where an emerging domain has new requirements or the project contains very specific demands. A *custom* system that precisely satisfies the needs of these projects may command a value that justifies its construction.

The present model of computing favours the approach in which operating systems are considered purely as a target for execution rather than an element of system design. This is partly because actually analysing or modifying the operating system to work more closely with the application being created can be extremely challenging. Systems software is written by a relatively small population of specialists. It is too much to assume that such a specialist in the specific system being considered will be present in any given team, a possibility that decreases as the complexity and variety of systems software continues to grow. The time to train someone to the point where they can modify systems software, and to train enough people that the implementation can be constructed in the time available to the project, can quickly become impractical. There is also the case that much of the systems software is proprietary and source access is not possible. Not only can such a system not be modified, but it may not even be clear how its operations will interact with the project being considered. In the worst case, a project team may not even know that there is a significant interaction until they discover it experimentally during implementation.

The result is that even projects which could gain from including a customised operating system as part of the delivered product might not be able to estimate how much of an advantage it would give. The cost of modifying the operating system to discover this information, or even to take advantage of a known optimisation possibility, can often prove to be unviable in practice. The potential or possible benefit is outweighed by the estimated costs. This also becomes a

self-reinforcing phenomenon. The perception of systems as inflexible means they are less likely to be considered as elements in a project, which means that expertise in systems will not be developed within the software group. Likewise, system software implementers, seeing little demand for flexibility from application software developers, focus on internal concerns within the existing systems.

In the current model the development team might not even consider systems software on the assumption that any work in that area will consume too many resources to possibly be worthwhile. If a team member is asked, "How will the systems software affect our project?" they are likely to find it a hard question to answer unless they have relatively specialised skills. Being told "Find ways to fix those systems effects that are disadvantageous to our project" is a monumental task unless multiple people possess extremely rare skills and experience. Substantial modifications or reconfigurations of system software are not easy tasks to analyse, estimate or implement. One goal of this thesis is to make it easier for a software professional to craft a reasoned and logical response to enquiries such as these.

If analysis is performed it is expected that a common outcome will be that there is marginal opportunity for operating system optimisation. In many cases the needs of a project are sufficiently general or non-demanding such that they are fully met by a general, off-the-shelf, operating system. In such a case it is assumed that any possible improvements at the system level will not provide a meaningful and valuable advantage to the users of the system sufficient to justify the cost of discovering and correcting them. As such it is an estimation of cost versus benefit that could be changed by a better ability to perform analysis or modify the system software but which will result in the same outcome for many projects.

The opportunity for new system construction is enhanced by discovering projects that have specific needs that are not being met by existing systems. This creates potential value for a system that can provide the needed capabilities. The possibility of a fully customised system may also suggest other optimisations when the system software is built around the project goals. This value provides an incentive which may justify the effort involved in the creation of a custom system. The projects for which a custom system is economical is likely to be a small segment of the total system market but it may be under represented. If system software is considered inflexible then such opportunities will simply not be identified or analysed and the opportunity will be missed.

1.2 Computing Systems

> "Just as a good manager faces the problem that his employees are unaware of his management, so does the systems designer suffer because the better his system does its job, the less its users know of its existence"
>
> - *Gerald M. Weinberg* [Weinberg, 1971]

Another perspective is that of the operating system designer. This would involve a project in which construction of system software as a product is the focus. This thesis uses the wider scope of computing systems, which includes all components required for software to execute. The best known as independent products in their own right are operating systems. An operating system is a software product which, according to one of the most popular general texts [Tanenbaum, 2001] provides two services and has one distinctive property:

1. The first service is the creation of an abstract representation of the underlying hardware.
2. The second service is arbitrating cases of resources contention between users of the system.
3. The distinctive property is that an operating systems runs in 'kernel' mode, a hardware mechanism that protects it against interference from non-system code.

This definition is incomplete, especially in relation to system design. It postulates an operating system outward view, in which the system is an independent element that software must call upon. It encourages the vision of an operating system as a static construct that is installed and goes about its task well before application issues are considered. This view partitions the computer into an operating system and the software elements that use the services it provides. Each element of the system is made easier to understand through having a clearly defined area of responsibility. Application programming concerns do not leak through to the system environment.

The reason for this view is that the programs that will be run on the system cannot be known at the time of construction. The designer knows that every design trade-off will favour some operations over others. This will interact with the behaviour of application programs either positively or negatively depending on what operating system operations they depend

upon. Attempting to optimise against all possible uses of the system is impossibly complex so the focus is on providing a concise, clear and reasonably efficient set of system functions and relying on the application to optimise itself if performance is an issue.

If custom systems are being considered then new opportunities emerge. The value of the system is determined by how well all elements of the system combine to meet the design goals of the project. This also means that the system is constructed with some knowledge of its intended focus, the applications it will run and the sort of operations that will have high value to the system. A general system cannot make an optimisation that could potentially cause negative interactions with some possible usage. A custom system is constructed around a design goal, a known set of application demands and developed alongside the applications. This allows a range of possibilities where system needs can be negotiated in a more complete way.

The 'general' operating system view is dominant. The idea of an operating system as a constructed response to local system demands is relatively rare, even in the research domain. The practice of computing is dominated by extremely evolved, long-lived, and widely deployed systems which must inherently be general systems. Indeed it is a rarely expressed but common belief that these systems have reached a degree of maturity such that there is little demand for innovation[1].

It is more likely that the possibilities for system adaptation and innovation seem insufficiently rewarding to justify the construction of new systems. This suggests that inertia due to complexity is actually blocking opportunities for potential developments. It should be noted that this disincentive decreases as the complexity of construction is reduced meaning that the possibility of exploiting a local optimisation becomes more attractive. As previous operating system constructors have observed [Brooks, 1995] there is unlikely to be a solution that makes systems construction trivial. Adopting a modular construction approach may be one step in making construction and adaptation of systems for a specific environment more feasible.

There is, specifically, a single observation that acted as a starting point for this investigation, which is that the domination of general operating systems has shaped the environment in which operating systems are developed. Commercial operating systems, of which Unix is the most

[1]Rob Pike's polemic "Systems Software Research is irrelevant" [Pike, 2000] is sometimes quoted in support of this position. However reading deeper indicates that the title is tongue in cheek. The paper itself states that research institutions have failed to generate innovation, rather than that there is none to be found. Indeed the slides from that talk and the essence of this thesis correlate strongly.

well known example from a research point of view[2], have been so successful and long lived that they have become a model for what an operating system is. Once accepted as a model all improvements within the field are most logically expressed within the context that system provides.

The result of this is that operating system development is broken up into relatively isolated strands of development. The model of an operating system that the Unix environment provides, being more precise and powerful than any abstract model (which are generally considered of value only as educational tools) actually becomes the environment in which development is performed. Instead of working within the field of operating systems the developer works within the field of Unix-like operating systems (or perhaps even a specific Unix). This can be seen in the fact that most operating system texts are tied to a particular system[Bach, 1987], while the general texts have sizeable case studies. A current operating system textbook[Silberschatz et al., 2013] includes detailed studies of Linux and Windows 7. The book used to include Mach[Rashid et al., 1989] but this is now relegated to an appendix. Texts that compare and unify disparate systems are extremely rare. It is simply more productive for the developer to express his work within the established context a specific system provides.

The disadvantage of this approach is that it creates a vacuum for those looking to develop novel, customised systems. There is a lack of models, literature, tools and reusable components all of which might make constructing new systems easier. The subsystems that Unix provides are so dependent upon one another, so linked to the design philosophy of Unix and the constraints it is based on, that they are not readily reusable in a truly novel system. The development of such a system is instead forced to start from very primitive foundations while being in competition with extremely mature and established systems. In such conditions, the viability of alternative system development is substantially reduced.

A modular approach to the design of systems has the potential to resolve these problems. It encourages a conceptual distance between a mechanism and the system in which it is being used, meaning its boundaries are more clearly defined. This modular approach allows commonalities between systems to be made explicit through direct reuse of the same implementation. The comparison of modules used focuses attention on what is unique about each system and what

[2]Unix is, at least partly, the dominant research operating system because its internal structure is both open and understood. The major competitor, Windows, is inherently isolated by commercial pressures at a substantial cost to the theoretical field of operating system development.

is shared between many systems. New innovations or improvements can either be integrated into an existing module or used to create a new module to best demonstrate the capabilities the innovation provides. Most importantly a mature modular approach can give the designer of a novel system proven modules and architectures using those modules with which to jumpstart the process of development.

1.3 Risks in Development

"Operating Systems are like underwear, nobody really wants to look at them."

- *Bill Joy, Chief Scientist of Sun Microsystems* [Economist, 2002]

There is a certain degree of paradox in looking at the field of operating systems. It is eminently clear that they are extremely valuable, complex and important pieces of software. They are a core component in virtually all computer related activities and the largest operating system vendors have invested vast sums of money in their research and development to produce some of the most complex software in existence. One of the most prominent operating system vendors has managed to make its product so prevalent that, in view of the importance of this software, many governments are greatly concerned at the power they wield [USvMS]. The same company even stated in public that interruption in its business would cause significant damage to the American economy[3].

The element of paradox comes from the fact that these incredibly important systems are, at the same time, not widely seen as interesting or rewarding topics for investigation. It is rare to even find the matter addressed directly, instead one must look for the unexpected gaps. As an example, a well known software engineering text [Pressman, 1992] dedicates a chapter towards programming language selection but no space whatsoever to operating system selection (even in the context of API[4] selection). There exist on the web a plethora of sites eagerly following the latest advancements in PC hardware to a dizzying degree of detail. These sites are often concerned with, fundamentally, quite small differences in behaviour and performance between

[3] "Breaking up this company would be a punitive proposal that would fundamentally harm consumers, the industry and the American economy." [Online, 2000]

[4] An API, or Applications Programming Interface, connects the programming language to the operations provided by the system being considered. In the context of Operating Systems it determines what system functions a program has access to.

competing products. Yet there is less focus on comparing operating systems in the same manner, even though operating systems make up a larger percentage of a domestic computers purchase price than ever before[5]. This value is often understated because the operating systems are heavily discounted for bundling with a new computer through the use of an original equipment manufacturer (OEM) license. To the purchaser the operating system appears to be "free" and implies the system is designed to work best with the included operating system[6].

The situation in the computer science field is not significantly better. The dominant focus of higher study remains the Unix operating system which now has a 40 plus year history and is highly evolved, mature and well established. A degree in computer science, such as the one proposed by the joint task force on computing curricula [JTF:2013], is already under pressure because the number of areas needing coverage has grown while teaching hours have not. The operating system area itself, to which the study recommends from 4 to 15 hours of coverage, is recognised as having seen significant growth in complexity. The end result is that the material recommended must be abstracted to a fairly high level in order to fit within the time available. The graduating student will almost certainly not have examined real operating system code. Translating the high level model they have been taught to a usable understanding of real operating system mechanics represents a substantial task. Nor is it likely that there exists a large number of ongoing projects, or experienced system architects, ready to provide further development after graduation. A poignant summary, possibly unwittingly, is the following quote:

> "Creating an operating system, like creating a computer, is an opportunity few engineers ever get. Most operating system engineers spend their entire careers enhancing or modifying existing operating systems or designing new ones that are never built or are never marketed... Those systems that are completed often don't catch on in the marketplace or are largely irrelevant because existing applications require the old system to be supported throughout eternity."
>
> - *Helen Custer* [Custer, 1993]

The solution to this paradox is that there are two balancing elements when considering the

[5] "We have increased our prices over the last 10 years [while] other component prices have come down and continue to come down." - Microsoft Senior VP Joachim Kempin [Kempin, 1998]

[6] One of the findings from the anti-trust case against Microsoft [DoJ:1999] was that their OEM agreements included a substantial price penalty if systems were sold without a bundled copy of the Windows operating system.

construction of an operating system. As with all large scale software projects any reward that motivates development comes with a risk that the project will not run smoothly. The risk in operating systems, that the project will fail to produce a viable outcome, has been well demonstrated but not necessarily documented, both in the lore and the literature of computer science. The Multics project, one of the first efforts that was specifically focused on the construction of a general operating system, was ultimately a complete failure with "no visible result" [Salus, 1994]. IBM's construction of OS/360, notable as being the first time an operating system became a competitive advantage rather than a required component, absorbed so much money and time that it put the largest company of the day at significant financial risk [Shurkin, 1996] even though it was ultimately successful. It was quickly recognised that operating systems are extremely complex constructs. This complexity is a source of project risk if time pressure is a factor in the construction of the system, which is generally the case.

> "The amount of effort required to write UNIX, while not inconsiderable in itself (10 man years up to the release of the level six system) is insignificant when compared to other systems. (For instance, by 1968, OS/360 was reputed to have consumed more than five man millennia and TSS/360, another IBM operating system, more than one man millennium.)
>
> - *John Lions* [Lions, 1996]

This estimated risk is balanced against the return expected on completion of the new system. The quote from John Lions, which indicates the expenditure of substantial resources, supports an inertia against replacing the system, a situation reinforced by the need for software compatibility. This provides a substantial disincentive towards any significant change. An operating system development proposal must argue convincingly that the substantial investment in effort and time together with the risk of failure will generate a sufficiently valuable benefit at its conclusion. This is possible in some cases, such as in the case of an entirely novel environment, or replacing a system that does not work, but making the case against a system that is functioning, even potentially sub-optimally, is much harder. It can be difficult to prove that a problem exists, that the new system will offer substantial advantages or that either of these cases has sufficient depth and scope it cannot be worked around. In essence the argument in favour of the status quo is much easier to construct and defend.

The standard tool for this sort of analysis is benchmarking, in which the two systems are measured over one or more parameters considered most relevant to their performance. This can give concrete information about the performance of an existing system, and thus the potential need for a replacement. It could also allow an estimate as to whether some internal operation is not making optimal use of the underlying hardware facilities. This estimation is excellent for refinement of an architecture. It is less capable of reliable estimation when the proposed replacement includes substantial change in the architecture itself. Low level operations (such as those measured in [Liedtke, 1995a]) will not change, but their context, operation and contribution to the overall performance of the system will.

The risk of constructing a novel system and the difficulty of proving potential advantage have a common root. That is, the systems are highly integrated both in design and construction. They are built from the ground up and the design goal and optimisation process will ensure a tight coupling of the parts from which they are constructed. The design discipline of operating systems has some reasonably general mechanisms, some highly evolved implementations, and very little in-between the two.

In such an environment it is hard to experiment with alternatives. Building a new system is extremely expensive and carries a substantial risk that a design concept will not pay off. Modifying an existing system means working against the system wide assumptions and coupling built into every component. Proving that a design goal is valid and generates superior behaviour is very difficult without a working system. Even describing a system that is substantially different from the reference models will require so much detail as to become impossible to absorb or be so abstract its implementation will be unclear.

One solution is to use a very loosely coupled system in which each component makes few assumptions about structures or other components in the system. Interactions will be simple, communication will involve passing data structures and much of the detail will be negotiated at run-time. The problem is that such a system will pay a performance penalty and thus not accurately represent the real world systems that will be unwilling to integrate or accept any performance penalty.

A better solution is the same one suggested for enabling system construction. A modular system is a much more practical conceptual model of a system. It is assumed that the modules will be processed such that they are optimised and integrated in the product. However, the

concept and structure of the system is easier to model and understand at the level of abstraction provided by the modules. This is similar to the way in which a programming language is a good way to understand code even if it is not an exact representation of what will execute inside the CPU. The model as a concept, and the model as it is integrated into the system, is connected by an automated and reproducible process that can be considered separately.

It is also possible to collect functional similarities in an even more abstract form. It is probable that every operating system will face the general issue of memory management. Looking at how a variety of systems implement this might allow common elements and specific optimisations to be determined. Common elements could possibly be made generic and thus eminently reusable. Specific elements can be tested and understood for the advantages they bring and whether there are subsets of commonality. In short investigation and experimentation can be performed on the subsystem level. These subsystems are also usable, and reusable, in real systems as the process to integrate them into a specific and task optimised system is considered a separate step.

1.4 Thesis Goals

> "The problem lies in the rigidity of our machines and, through them, in the rigidity of our programming languages. Whenever a man is confronted with a new machine, he is forced to choose between making some adjustments in himself or adjusting the machine to narrow the gap between what is desired and what exists. Although machines, and especially computers, are adjustable, the time scale for them to be changed is generally much longer than for a person."
>
> - *Gerald M. Weinberg* [Weinberg, 1971]

The problem being addressed is not that existing operating systems are flawed and a new system is needed to repair that flaw. The argument is that existing systems are too successful, and the field's abstract models and component toolbox so weak, that it is difficult to consider constructing alternatives. In doing so there may be possibilities for system optimisations that are not being exploited. The thesis considers a process that may allow systems software to be discussed, considered and constructed in a new way.

The first goal is to provide the framework for a modular approach to operating systems. This is one of the principal mechanisms of abstraction that allow large systems to be efficiently constructed and components reused.

The second goal is that the modules must be efficient in operation. Application software tends to be either relatively parallelizable or have reasonably soft performance constraints. System software derives much of its value from the efficiency of its performance. A module system that does not allow optimisation or requires run-time overhead is generally not acceptable to the end-user. The fact that the system is constructed in a modular fashion is not a benefit to the end user. The possibility of the deployed system providing lower performance is strongly undesirable to the end user.

There are some other "soft" goals that it will be impossible to prove or fully explore in this document. However, they are useful as indications of future paths towards a mature system development process. The first of these possibilities is being able to provide a practical model of the systems domain through a rich collection of modules and systems constructed using these modules. The second is allowing iterative development in which projects can be constructed in terms of advancing or modifying a module or structure of modules rather than building an entirely new system. Together these allow advancement in the field of systems research to be performed at a smaller and more practical scale while capturing any advances made for future reuse. To design all possible pieces is an impracticably huge undertaking. To design a process in which pieces can be constructed and built into larger structures may provide a framework in which many hands and minds could move towards such a complete system.

One risk should also be mentioned. For any project such as this there is a desire to build a perfect system, an ideal component or a universal system that can be adapted to any challenge. These are immensely tempting objectives from a technical challenge point of view but they are also traps. The problem can be neatly stated in that a system that is "perfect" or "ideal" for one context is unlikely to be perfect for all other possible contexts. Attempts to be "universal" become so complex and unwieldy that they are never completed or have an immense conceptual and operational complexity that makes them impractical and inefficient. Significant complexity can be integrated into the act of selecting an appropriate component from a pool of possibilities and making that component clever in its automated self-optimisation. The focus on process, iteration and a modular approach reduces the temptation to try and produce complete or perfect

systems.

Making the operating system an integral element of a whole system solution, rather than an inflexible foundation, means it is easier to see potential gains that can be realised. If an application, hardware platform or operational environment has specific needs, it may well be that the operating system is the best place to address them. If the system environment has specific constraints, the operating system can be compared against those constraints for conflicts and potential optimisations. It opens up the possibility of the field of operating systems, which is dominated by established "one size fits all" solutions, becoming a more active participant in crafting new systems and capabilities.

The thesis develops a software design methodology referred to as Shards to demonstrate how this approach could be structured. Shards provides a framework to support the encapsulation of operating system functionality in a modular form so that system construction is easier. Shards provides mechanisms by which the goals and structure of the system can be captured in a form suitable for automation. These are joined together by an automated process in which the modules are able to respond to this information in order to select and optimise themselves prior to the process of compilation into an executable system. The modular model of the system and the unified implementation are given a degree of separation in this process. This allows the modules to focus on global concepts and reusability while the implementation can focus on raw efficiency. There is not a one-to-one mapping between the two, and the process is effectively one way (information is lost in the process of construction), as is the case with any compilation. However it will be reliably reproducible which is sufficient.

This approach is unique among those found in the survey of the operating system literature. The vast majority of operating system work is innovation in fundamental mechanisms or iteration on specific operating system implementations. There has been work on code generation, optimising compilers and transform-oriented languages which have some degree of similarity in the mechanism of operation. Indeed the Shards process could be used for the development of general code[7].

The primary focus is to demonstrate how such a methodology could be implemented and consider if the concept is viable. Since the domain being addressed is extremely large, some bounds must be put on the scope of the project. This thesis does not seek to produce a new

[7]The system has been used to auto-generate an automatically cross-linked web site from a non-HTML data file.

operating system, new operating system functionality or an operating system optimised for some specific environment. It is a process rather than a product. For the same reason it does not focus on the production of mature operating systems modules (the development of any module would easily be a thesis in itself), a complete set of modules or even the ideal structure to support the goals. It is expected that substantial iteration would be required in any attempt to provide a full system.

1.5 Thesis Outline

In Chapter 2, an overview of the development of computing systems is provided. Once again it avoids focusing on specific systems and is more concerned with the processes that created them. Many of the assumptions incorporated in the design of operating systems, and their role, live on long after the environment that created them has passed into history.

In Chapter 3, some of the existing systems within the domain are explored. Statistically the number of operating systems is relatively small, with a handful of successful systems dominating the field. This gives the appearance of a unified environment. However when considered by design focus, which reflects that the computing environment is not uniform, it can be seen that there are a great many specific demands which will shape how systems are constructed. This also opens possibilities for the discussion of systems built around a specific design focus.

In Chapter 4, attention turns to the form an optimised operating system could take. It is argued there cannot be a single optimal implementation for all systems and design goals. Instead, the evolution towards a more general process of design and construction of optimised systems is introduced.

In Chapter 5, the design for an automated mechanism is presented. This tool, named the *Shards* system, aims to provide a mechanism for capturing system components for use at the design level. These elements gain the advantages of modular software and provide a way of capturing operating system theory in a rigorous form that can be applied to simplify the process of system construction.

In Chapter 6, systems with some similarity to Shards are discussed. The essential mechanisms that Shards depends on are not unique in the field of computer science. However, the application of these mechanisms, required for the goal the Shards project set itself, is substantially different.

Chapter 7, considers the Shards process which is both an implementation of the ideas de-

veloped in Chapter 5 and a requirement for the approach being proposed. The Shards system aims to control complexity, reduce run-time performance costs, and enable experimentation and iterative development, all of which require a more dynamic approach to system construction.

Chapters 8 and 9 seek to explore the application of the implementation presented in Chapter 7. A single area within the operating system domain is covered in sufficient detail to explore the environment in which Shards will be applied. This leads to some simple examples of how a technical environment gives rise to optimisation possibilities which provide the motivation for the development of Shards components.

In Chapter 10 a conclusion is presented discussing how practical this approach proved to be, lessons learned and ideas for further work that could be carried out.

Chapter 2

The Development of Computing Systems

The history of computing is an interesting one. Science, working towards an understanding of the natural laws, began to find that application of this understanding required prodigious amounts of computation. It was not that the mathematics was innovative, or complex, just that there was too much of it. Computing was developed to fill this need and thus took on many of the qualities of Engineering as it was designed to apply science (such as the transistor) to a pre-existing problem. It was also developed to automate virtual concepts, such as mathematics and accounting, rather than take direct action. The operating system, which exists as a machine inside the machine, is even further removed from directly interacting with the outside world.

This situation gave a great deal of design freedom in how the computing system is constructed. However the goal was not to design the ultimate computer. Instead it was to efficiently create this complex tool so that it could be applied to important problems urgently needing a solution. As a result the focus was on fast and evolutionary development based on previous solutions. This means that most modern systems are connected to, and shaped by, a process of evolution that is longer than might be expected.

For this reason it is worth taking some time to examine the history of computing systems. It will be seen that over time the definition, role and even existence of operating systems changed. This is discussed in the "system elements" part of each section which considers the degree to which there was an internal process which managed the internal operation of the machine and provided additional services to the users of the machine.

2.1 Comment on Computing History

The heritage of operating systems can be seen in the history of computing though the records are limited. Computing on a wide scale has had extremely profound effects in an extremely short amount of time. A mere 100 years ago the most advanced computational device was a Burrough's mechanical calculator, whereas now computers of incredible power are commodity products. It would seem logical that such a revolution would have been carefully explored, documented and its artefacts recorded and archived in museums of computing. This is not the case however; computer history is a sadly undeveloped field and a massive amount of information has been lost, much of it irretrievably.

This general weakness is particularly acute in the field of operating systems. The context of a given system includes the intent behind creation, the practical usage, the practical flaws and a lot of other human elements that are rarely recorded. Unlike the actual machinery of computation, the hardware, few artefacts are left from which this environment can be reconstructed or analysed. Technical papers, where they exist, tend to focus on quite narrow and abstract innovations. This is partly in recognition of the fact that most computing systems are tied to a given hardware platform and frequently 'die' when that hardware becomes obsolete.

In short, solid information on many foundation systems, whose design innovations continue to shape the field, has been lost. Part of the reason for this is that the field does not really have the means to identify and isolate innovative components within the larger system. Innovations with a potentially wider scope or continuing applicability remain contained within their originating system and are lost when it ceases to be used. Documentation, because it would have to encompass the entire system to have any value, requires a massive investment of time and even then may miss the most interesting parts. A methodology that allows a system to be viewed as components, such as the one in this thesis, could have potentially alleviated this situation.

In practice at least some of the information survived, although it will never be possible to know how much did not. It is no secret that Unix was fundamentally shaped by the innovations developed for the Multics system, even though Multics itself was a practical failure. Likewise Microsoft's NT system was shaped, at least in part, by the Prism project at DEC (circa 1985-1988) even though that project was never released. The medium of transmission was not the documentation or the physical artefacts the earlier projects generated, but the knowledge within the mind of a skilled practitioner. Needless to say, information stored in this way is not ideal

for the development of a general field.

2.2 Mechanisation

Computers are, historically, a very recent innovation. However computing, in terms of forming procedural steps to complete tasks, has an extremely long history. It can easily be imagined that communication itself was shortly followed by someone giving someone else a list of orders to follow. Tanenbaum [Tanenbaum, 2001] uses the familiar example of a cooking recipe to demonstrate an ordered list of instructions that has a very long heritage. Recipes can also include such elements as conditionals (if brown, remove from the oven) that remain fundamental operators in modern programming. They differ mostly in that they are quite informal in their expression, as it is assumed that the person following the instructions is capable of making intelligent decisions in their application.

The field of mathematics, which demanded more precision and less error in its application, was far more rigorous. It was this human endeavour, which itself has a very long history, that was the origin of modern computing. Indeed our modern term for the instructions a computer follows, algorithms, is itself derived from a mathematics text dating from the year 830[1]. It was also a field where the calculations and precision required exceeded the convenient capability of the people required to make them. Thus structured methods, notation and memorisation aids were seen to have value. Examples of the tools developed to help include the well known Abacus and more advanced examples such as "Napier's Bones". These last were developed in 1617 by John Napier and were able to reduce the mathematical problem of multiplication to the much simpler problem of addition and subtraction.

Probably the highest point in the process of automating mathematical operations came in 1822 when Charles Babbage began to work on the Difference Engine[Shurkin, 1996]. This machine, the concept of which had actually been advanced separately by Mueller in 1786, would generate tables of polynomials by the method of differences. Since the manual construction of vital mathematical tables was consuming immense amounts of human effort, and the end result was highly error prone, there were sufficient practical applications to generate sizeable funding for the project. The project was one of immense complexity. The British Science Museum[2],

[1] Kitab al-jabr w'al-muqabala (Rules of restoration and reduction) written by Abu Ja'far Mohammed ibn Musa al-Khwarizmi (ca 780-845).

[2] http://www.sciencemuseum.org.uk

which holds many of the design documents, states that the plan called for 25,000 parts and would weigh 15 tonnes when complete. For various reasons, including Babbage's distraction by an even more advanced machine, the project was never actually finished.

A much simpler machine, but one that had a lot more influence on the development of automated calculation, was the 'Tabulator' built by Herman Hollerith in 1890. This machine was intended for a specific real-world task, specifically collating the data from the American census. However this task was symptomatic of a much wider growth in the volume of data that business and government were facing. The advantage of this task is that the calculations were relatively simple and the demand was continuous and rapidly increasing beyond human capabilities. The ability to better understand the data being generated represented a potential competitive advantage. As such the economic incentives available to fund development of the business, which eventually became one of the foundations of the company now known as IBM, were substantial. This machine also represents the start of business computing, which for much of the early computer history was firmly divided from scientific computing.

2.2.1 System Elements

The machines of this mechanical computing period were, unsurprisingly, dominated by the engineering complexity involved in their construction. There was not sufficient flexibility to do much more than implement a single, very specific, algorithm. Any piece of the engine that was not vital in the operation of the machine represented substantial engineering for no gain. Even interaction with the user was secondary to supporting the mechanisms of the solution, for example needing to manually adjust components within the machine to set a starting value.

The downside of these machines was the cost of construction and the inflexibility of the final product. This meant that only a narrow range of calculations could be automated, and even fewer of those had sufficient commercial value to support the costs required for development. The machines were also highly integrated, so it is relatively unlikely that anything other than the most basic components of one machine could be reused in the next. This high cost of construction meant that good design was vital. Thus the mechanisms used in calculation, the algorithm being represented and the system elements that would make them usable, were entirely unified during the design stages in the interests of structural simplicity.

2.3 Operation Abstraction

The calculating machines of the previous section were too expensive and took too much time to construct for a whole range of problems to be viably addressed. The logical progression was to observe that many of the operations, which were after all derived from mature mathematical methods, would be common to many solutions. If the application of these mechanisms was flexible then the component and possibly even the whole machine could be used for multiple tasks. This would allow the costs of construction to be amortised over multiple applications, although this flexibility would have to be balanced against the increased complexity required to support it.

Babbage, during the many years involved in the construction of the difference engine, realised this and began drawing up plans for a flexible machine called the Analytical Engine. Considering the complexity of the difference engine, and the efforts its construction was absorbing, it is uncertain whether he truly believed the machine was likely to be practical. The design, which evolved over many years until his death, was an impressive feat which predicted many of the advances on which modern computers depend.

His machine included input in the form of punched cards, which had been developed earlier for the automation of weaving, and output in the form of a printer, curve plotter, card punch and a bell for the operator's attention. The plan called for a general purpose arithmetic unit (the mill), a substantial store for variables, and instructions to move data between the two. The programming operations included conditional execution and loop constructs. Considering the period at which he was working it was an impressive feat of design.

It was also relatively well documented, Babbage being a famous individual of the age and others being impressed by the possibilities his vision offered. The best known of these was Ada King (nee. Augusta), Countess of Lovelace, who was an excellent mathematician. Her extensive annotation of a translation upon the subject of the Analytical Engine [Augusta and Menabrea, 1842] reveals a deep understanding of the capabilities the design offered. She is generally credited with being the first programmer for her work, over many years of discussion with Babbage, on how to program the machine for various tasks. Sadly the fact that the machine was never constructed meant that its advances were never truly demonstrated and it did not have significant influence on later developments.

In 1936 Alan M. Turing, who was aware of Babbage's work, created a logical model for a

virtual computer as part of a paper on the subject of computable numbers [Turing, 1937]. The model was extremely elegant because it was not defined by the limits of engineering concerns. It advanced the concept that program instructions were simply a specialised form of data, as well as the use of state transitions to represent computation. Turing reasoned that the actions of this machine were capable of representing any mathematical operation that a human could do manually. Turing was involved, along with many others, in the wartime construction of decryption machinery used to decode German communications. Since the work done and machines produced remained a national secret until 1970 it had minimal influence on other developments in the field. However, Turing (along with von Neumann) is often credited with the theoretical foundations for general purpose computers[Silberschatz et al., 2013].

The last computer for this section is the one that did have substantial influence even though it was not the first electronic computer. The ENIAC [Eckert, 1988], finished in 1945, became one of the first retargetable machines to be used on practical applications. The publicity surrounding it had an important effect on the recognition of advances in computing and the advantages of digital circuitry in the construction. At the same time a detailed report on its planned successor [von Neumann, 1982] disseminated the technological lessons learned in its construction, much to the chagrin of the creators who were not credited in the first, widely distributed, draft. The ENIAC was not a stored program computer but it was sufficiently flexible to be retargeted to a wide variety of applications.

2.3.1 System Elements

The machines discussed above, although only ENIAC is a fully operational example, had no concept of an operating system. As has been mentioned they were retargetable rather than programmable. The configuration of the machine determined the capabilities and sequence of operations the machine was capable of. This configuration did not change during a run, and indeed required the machinery to not be in operation while changes were made. The specific mechanism, in the ENIAC, was through plug-boards which allowed the connections within the machine to be modified and switches which determined what operation each functional unit would perform.

Essentially the systems all had two modes. The operational mode proceeded in exactly the same fashion as the earlier mechanised computers. The structure of the computer was a

direct expression of the algorithm being implemented, and every operation within the machinery moved the computation forward. Since there was no flexibility, and precious little capacity, at run time there was no real role for an operating system. The complexity of the hardware was also dominant, meaning that interaction with the machine was primarily determined by whatever was most convenient for the hardware.

ENIAC's other mode introduced the concept of programming. It was not a trivial intellectual task to determine how an algorithm could be expressed as a configuration of the machine. The individuals who carried out this task had to effectively invent a new field. Since their programs were vital in determining how flexible the hardware could actually be their role in the ENIAC program was central. In addition, since the implementation of their programs and resolution of any faults required the valuable machine to be out of operation, the pressure was no doubt substantial.

> They had no users guide. There were no operating systems or computer languages, just hardware and human logic. "The ENIAC," says Ms. Bartik, now 71, "was a son of a bitch to program." [Petzinger, 1996a]

It also marked one of the first examples of 'soft' computing being poorly represented in the history and largely undocumented. There is little doubt that the hardware, with its impressive dimensions and blinking lights, was the centre of attention. The leaders of the project were also hardware designers and thus would have focused on their interest. The ENIAC in practical usage tended to run one application for a substantial amount of time to minimise the down-time re-wiring represented. It has also been suggested that the under-representation can be attributed to the fact that the programming team was entirely female [Petzinger, 1996a;b] and were officially considered "sub-professionals". Regardless of the reason the role of programming in the ENIAC project does not feature in the literature, and has left behind few methods or artefacts.

It is almost certain that the programmers did develop a range of techniques to help them program the machine. Since they had limited access to the machine[3], and it was no help in making their task easier, these techniques would have all been either on paper or within the minds of the team. These techniques would have focused on commonalities in implementation, allowing

[3]As sub-professionals they were unable to gain sufficient security authorisation to access the computer for which they were writing programs.

them to both communicate more easily and use lessons learnt in future designs. While their processes at this time are, as mentioned, undocumented at least two of the ENIAC programmers (Jean Bartik and Frances "Betty" Holberton being the best documented) went on to make important direct contributions to computing. A quote from Betty Holberton, who later worked on C-10, COBOL and Fortran, expresses some of the difficulties of being a programmer at this time.

> "I spent half the day trying to figure out what people needed in computers and the rest of the day trying to convince an engineer it was his idea."
>
> Betty Holberton [Petzinger, 1996b]

It is reasonable to suggest that the programmers of the machine would have built methods and systems to make the computer more practically usable. These systems probably consisted of mental models, notations and reusable components although the details have not survived. In short they were, at this point, human systems. It was a natural reaction to the growing complexity of the task with which they were faced.

2.4 Early Systems

The initial computers, and there is a long string of them, are unified in one important design consideration. They were one-off machines designed for a particular task which, by today's standards, was extremely limited. They were also primarily interesting, to their designers, as experiments in how computers could be constructed. The idea of computers becoming not only multiple orders of magnitude more powerful, but at the same time cheap enough to be commodity products, must have seemed outlandish, a mind-set now eternally captured in Thomas Watson Sr.'s quote, "I think there is a world market for about five computers" [Salus, 1994].

It did not take many years before people began to realise the potential of the computer. Not only was it growing rapidly more powerful, and cheaper, but it also became clear that the range of tasks to which it could be economically applied was growing as a function of this development. The end result was that there was the potential of substantial wealth, and even fame, for the first to tap this newly created market. This led to some conflicts about the source of inventions and the justification of patent rights in the commercialisation of computers after the war[Shurkin, 1996] that complicated the recording of this period of computing history.

The ENIAC was the machine that proved the practical viability of computing, and it was from this base that the next developments would spring. The most direct line of descent was the two principal creators, Presper Eckert and John Mauchley, realising the potential for sizeable financial rewards if they could commercialise the computer. They founded their own company and began work on a far more advanced computer which would eventually ship as the UNIVAC[4], one of the first commercially available machines. They underestimated the cost and complexity of construction however, and by the time the machine was complete they had sold most of the company (to Remington Rand) to fund it.

The other line of development came from the commercial naivete of the University of Pennsylvania which over-saw the development of the ENIAC and which was ineffectual in claiming ownership of the intellectual property (IP) that resulted from it. Even worse, John von Neumann released a draft document [von Neumann, 1982], which did not carry the signatures of the ENIAC leads, which discussed the general design theory for the next generation computer (the EDVAC) being planned. He also released a later document [Burks et al., 1982] which expressed further thoughts on how the capabilities of these new machines could be harnessed. These documents propagated widely which, combined with the weak IP situation, led to it having a substantial impact on the field of computer design, something that was of little comfort to Eckert and Mauchley who realised that much of their theoretical lead had been lost.

Regardless of the human element that surrounded it the UNIVAC was an important machine. Not only was it powerful, and technically advanced, but it was also advertised at a time when computing was still a specialised and mysterious field. Since it was targeted at an audience whose specific needs could not be known, since the creators did not want to limit the people to whom they could sell, it was forced to consider the problems of general computing and even the education of programmers. It also had the advantage of the unquestioned leader in office automation, the industry leader IBM, having no competing product for many years.

The statistics of the UNIVAC [Shurkin, 1996] itself shows the rapid growth in computing power when compared to the ENIAC. It had 1000 words of memory, stored in mercury delay lines, capable of storing 12 digits. It ran at a speed of 2.25 million cycles a second and was elegant at a mere 14.5 x 7.5 x 9 feet in size. I/O with the machine was via magnetic tapes, with a huge capacity compared to card systems, which could be written on a UNITYPER and

[4]from UNIVersal Automatic Computer. The name gives a strong indication as to the scope and generality of their plans.

printed on a UNIPRINTER, although it is mentioned [Norberg, 2003] that the usability of the I/O peripherals was less than optimal.

2.4.1 System Elements

While a large part of the focus was on hardware, which was both advancing at a furious rate and providing a large number of interesting technical challenges, commercialisation drove interest in software. The population of skilled programmers was minuscule, effectively clustered around the quite small number of operational computers, and their knowledge was strongly tied to the specific machine with which they worked. Indeed the theoretical questions of how programs could best be expressed still contained many unresolved issues. EMCC (the Eckert and Mauchly computer corporation) was itself fortunate in being able to poach programmers from their previous projects, gaining both Betty Holberton and Jean Bartik, important resources that are rarely mentioned in the hardware-centred histories.

Thankfully the development of stored program computers meant that the physical effort of re-configuring the ENIAC was a thing of the past. This was required as commercial machines were unlikely to be dedicated to a single massive task in the same manner as ENIAC was. In addition the ENIAC design, which contained a great deal of parallelism, was simplified as a result of the observation that harnessing those capabilities made the construction of software forbiddingly complex.

The operating system, as an independent entity, did not exist. The systems were still sufficiently performance limited that all run time resources had to be devoted simply to solving the task at hand. An interesting insight is provided by observing a program that von Neumann wrote as a demonstration of how sorting could be efficiently programmed on the EDVAC [Knuth, 1970]. The most immediate observation is how limited both hardware, application problems and programming languages were. It is easy to take for granted the development and insight that our modern languages represent the culmination of, and the expressive capabilities they represent.

In the von Neumann document, it can be observed that the compilers involved in the programming languages of this time are what we would call an assembler today. There is a one to one correlation between instructions and machine operations. The process of assembly itself is suggested through mechanical translation, a specialised typewriter in which each language token would be a key that generated its machine encoding. Structure within the language is relatively

simple, partly as a result of the fact that it would be the product of a single mind. The program also contains a number of programming imperfections, indication that it was challenging even for a mathematician as capable as von Neumann.

Far more interesting is that operating system elements can be seen to exist within the program, and even account for a reasonable percentage of its size. The computer being programmed had a mercury delay line store, a mechanism in which pulses could be stored within the delay inherent in signal propagation through the medium. This also means that the signal is only available for use at the same interval, when it is picked up at one end and re-transmitted at the other. To reduce this delay the document mentions short lines which hold only a single word giving them a much shorter access interval. The system thus had three levels of memory; registers, short lines and long lines. The control of this memory was entirely under the control of the programmer, and the degree to which it synchronised with the operation of the program would have a substantial impact on efficiency. In simple terms system considerations existed, but their solution would depend on manual integration by the programmer.

This was not an issue for a genius like von Neumann doing a single program as an academic exercise. However it was a recognised limitation for EMCC, and others, for whom the generation of programs was a determining factor in how many computers they could sell. This was especially true because they wanted to sell these machines to corporations who rarely had established programming talent in house. Nor was there a significant pool of skilled practitioners seeking employment. The only option was to invest money and time in seeking to ease the process of software creation. There was some opposition to this idea though from programmers worried about losing efficiency and others who just didn't see it as that important a problem.

> "He (von Neumann) didn't see programming as a big problem. I think one of his major objections was you wouldn't know what you were getting with floating point calculations. You at least knew where trouble was with fixed point if there was trouble. But he wasn't sensitive to the issue of the cost of programming. He really felt that Fortran was a wasted effort."
>
> - *John Backus* [Shasha and Lazere, 1995]

The development of programming languages is a subject with a rich history in its own right. One aspect worth noting is that there is no concept of a difference between application and system

code. A program would have to contain and interleave both facets of programming to perform accurately and efficiently. The idea of how this could be ordered, let alone encapsulated into a higher level form, was far from obvious even though the development of high level languages seems obvious and inevitable now.

> "In order to perceive the real depth of this subject properly, we need to realise how long it took to develop the important concepts that we now regard as self-evident. These ideas were by no means obvious a priori, and many years of work by brilliant and dedicated people were necessary before the current state of knowledge was reached." [Knuth and Pardo, 1977]

One history of development [Norberg, 2003] is a good starting point since it is contained within a single corporate entity with a strong commercial interest in the subject. It makes it clear that there were several components developing in parallel. Firstly the programming team interacted with the hardware designers in what operations were needed and which could be done without, leading to the machine codes C1 to C10 developed within EMCC. The machine code was also quite simple by today's standards, the UNIVAC having a total of 45 opcodes, and the average program being reasonably small. A reasonable percentage of both opcodes and programming effort was devoted to buffering and parallelism in I/O to avoid the processor being idle.

Language effectively developed as a relatively natural response to recognised repetition in the activities being carried out. One avenue was to collect representative implementations of common programming elements. While the total range of programs is infinite the practical requirements of the tasks being programmed were much narrower. As a result certain routines were recognised as recurring which suggested that a previously written version could be reused, a process which would dramatically cut programming time. The UNIVAC machine shipped with a rich library of routines which purchasers could use in their own programs. The importance of this facility was recognised and formalised [Shell, 1959] by IBM for the 705 computer, which was its competition to the UNIVAC.

The other avenue was the expressive power, and ease of use, of the machine itself. The translation of symbols more convenient to the programmer into the far more cryptic machine code was one clear need. More advanced manifestations included automatically 'compiling' routines

into the code that used it, first expressed by Grace Hopper in her A-0 compiler [Hopper, 1987]. Another was recognising that one instruction, with parameters, could provide enough information to generate a sizeable amount of code. This technique was used to solve specific domain problems, one of the first examples being a sort generator written by Betty Holberton. Another example is the speedcoding system [Backus, 1954] which could expand and solve arithmetic expressions.

The many elements of a language, which today seem natural and inseparable, had a wide diversity until it was unified as a single coherent system. Unlike the task specific languages that excelled at one problem this language would be truly general, capable of expressing everything that could be expressed in a far more structured and convenient manner. The project lead John Backus [Backus, 1976] observed that the economics of computing were beginning to stress programmer productivity. In addition code inefficiency, which had been concealed behind the slowness of system library routines such as floating point arithmetic, was becoming more visible as these functions became integrated into the hardware. This meant that partial systems could not offer the productivity required while simplistic systems could not offer the performance necessary[5].

What may seem surprising is that there was substantial opposition to his plan. His recollections are coloured by images of a "priesthood" that was hostile to the popularising of programming. An even larger number felt that the act of programming was simply too complex to be automatically translated to efficient machine instructions. Even the team developing the language was not sure that their goal was achievable.

> "At that time, most programmers wrote symbolic machine instructions exclusively (some even used absolute octal or decimal instructions). Almost to a man, they firmly believed that any mechanical coding method would fail to apply that versatile ingenuity which each programmer felt he possessed and constantly needed in his work. Therefore, it was agreed, compilers could only turn out code which would be intolerably less efficient than human coding" [Backus and Heising, 1964]

The actual result is now clear to us. The language, known as FORTRAN for FORMula TRANslator, still has a direct modern descendant (The most recent language standard was For-

[5] Although Backus, modestly, also credits his "unusually lazy nature" as a prime motivator for programming automation.

tran 2008) in productive use. Perhaps more importantly from a system point of view FORTRAN formalised our expectations of what a language should look like, what it should contain, and that it should be a standardised structure so that it can encompass a wide variety of environments. If your program could be entirely expressed in FORTRAN then it would most likely run on any machine that contained a FORTRAN compiler. The actual specific hardware details, such as the bit size best suited to hold an integer value, is generally sufficiently abstracted by the compiler that the programmer could work in a machine independent manner. From a systems point of view this significantly altered the importance of system software. Shipping a computer without FORTRAN would be a crippling weakness, and implementing a novel operation not supported by the FORTRAN compilers would be less accessible and thus less valuable.

As a programming tool there was significant continued resistance to the concept of high level languages. Experienced programmers had serious concerns about the practical efficiency of high level languages, although other observers suggested they saw it as a challenge to their position and authority. However its adoption was rapid given the convenience it offered, the increasing amount of programming required and hardware resources available. Eventually studies [Ridgway, 1952] examining the relative efficiency, for example showing that a sample program took 880 minutes to solve manually and 48.5 minutes using pre-Fortran high level languages, supported this dominance. A side effect of high level languages was that they became standardised interfaces to the computer. They also allowed much larger programs to be viably constructed.

> "The advent of programming languages of this kind [FORTRAN] some nine years ago vastly enriched the art of programming. Before then a program containing 5,000 instructions was considered quite large, and only the most experienced or foolhardy programmers would attempt one"
>
> - *Christopher Strachey* [Strachey, 1966]

2.5 Integrated Systems

One of the main points in the previous sections has been that system concerns have always been present in computing. What changed is the point at which they were handled. Originally the programmer was responsible for understanding and assimilating all machine elements in any program they wrote. The growth of language, which had nothing that we would recognise as

a modern operating system involved, did not solve this complexity. However the compiler did because it represented a mapping between the language functions and the underlying hardware operations, and system concerns, that would implement them. Effectively this part of the compiler, frequently called the 'back-end' was able to automatically resolve many system concerns without specific instruction, or even awareness, from the programmer.

This automation was the concern of the programmers at the time. They knew the compiler did not actually produce instructions directly equivalent to what they wrote. Had the compiler not been able to generate functionally equivalent code for a given system environment then they would actually have been right. However in practice the predominance of high level languages proves that, in all but a limited number of specialised domains, their concerns were not significant. Any inefficiency in the translation was swamped by the convenience and productivity of construction using high level languages. The compilers were able to merge programmer demands and system requirements in a single executable output[6].

It could even be said that the introduction of compiled languages, which included a layer of abstraction, drastically increased software's scope. It allowed application software to grow dramatically in size, reusability and functionality. Innovations in hardware now had to seek support from the compiler authors to make them available. Innovations in languages would have to entreat application software to be ported to it. Binary only applications were even worse since they needed the interface to be a constant in order to function.

At this point most histories lapse into very human-oriented stories about the next developmental stage. However this thesis is better served by an abstracted version flowing from the previous sections. The expressive power of language dramatically increased the range, and depth, of problems that could be viably computed. This had the effect of increasing the demand on computing time but also meant the patterns of interaction between the user and the software was increasingly complex. This process pointed out the central weakness in the compiler as an abstraction layer.

The back-end of the compiler is able to map a program to a sequence of instructions for a specific machine. However the mapping is inherently exclusive, the compiler must assume that the entirety of the machine belongs to it for the duration of the run. Since the compiled binary has no way of knowing what other software exists on the system it has no way of reaching outside

[6]Although it is worth mentioning that the formalising of language structures was not ignored in the hardware domain, leading to hardware which extended features for the compiler's convenience.

of itself. Nor is the hardware, which takes a passive role with regard to software, any help. The answer was clear, create another layer of indirection between the hardware and the executable.

> "Any problem in computer science can be solved with another layer of indirection."
>
> - *David Wheeler*, Chief Programmer, EDSAC[7].

This was the birth of what we would recognise as a modern operating system. It is little surprise that programmers were extremely interested in having one. The advantages of interactive debugging, on programs that were growing dramatically in size and thus potential for errors, was alluring. Indeed the programming task, which might require many runs each of which solves perhaps only a single bug, was forced to be interactive even if the cycle time on a batch system could be measured in hours or days.

2.5.1 Shared Resources

> "Suspicion of computer manufacturers is nowhere greater than in the introduction of time-sharing systems. Old-timers are often heard muttering that time sharing is merely another scheme to introduce even more inefficiency into computing, so as to further line the pockets of the capitalists. Certainly time sharing, like other computer innovations, was undertaken on a large scale with no psychological investigation whatsoever. People thought it would work, or wanted to think it would work, so it was pushed onto the market and the battle began"
>
> - *Gerald M. Weinberg* [Weinberg, 1971]

It seems difficult to imagine that people would resist the introduction of time sharing, a facility that is automatically assumed on modern systems. However it provides an interesting insight into the software environment of the time. The computer was modelled as a production line, with a hopper of jobs being fed into it, running, and being replaced. The idea of using the machine interactively, of having another process getting in the way of the application, and perhaps even going so far as to interrupt its processing, must have seemed inordinately wasteful.

[7]The EDSAC was another computer in the line of development from the ENIAC, planned for after the EDVAC which von Neumann described [Burks et al., 1982].

The programmers had a different view: their interaction with the computer was interactive by virtue of the fact that the software they ran had bugs. They were also aware of the hidden capacity of the machine, and not so awed by its complexity that they considered it an unchangeable element. Perhaps most importantly they didn't mind risking both programmer time, and machine performance, on an interesting experiment.

The first operating system in the line being followed, which was certainly not the only line of development in the field, began at MIT in 1959. John McCarthy was convinced that time shared systems were both a logical and necessary innovation that deserved a wide, and even interactive, application. He wrote a memo [McCarthy, 1959] suggesting that the department's new computer receive the hardware modifications that would make such a system possible. Specifically the machine needed a structure such that running programs could be *interrupted* and control passed to the operating system, a term which has been retained.

A formal, and externally funded, development process was begun to write a time sharing system. This project remains nameless in the history books as the project was never brought to completion. Instead, in a pattern that seems familiar in the field of operating systems, it was supplanted by a smaller, less ambitious, but more focused system that was sufficient for the job. Even at this point practicality surpassed theory. The system was called the Compatible Time Sharing System (CTSS) because it worked in cooperation with a batch system. It was built by a team led by Fernando Corbato, was capable of running 30 terminals, and was featured in a popular science magazine [Fano and Corbato, 1966]. It also proved that such an environment offered the possibility of new software (including runoff and typeset for document preparation) and new collaborative environments (such as inter-machine e-mail). It was this potential that most motivated the continuation of research into operating systems.

> "If computers of the kind I have advocated become the computers of the future, then computing may someday be organised as a public utility just as the telephone system is a public utility...The computer utility could become the basis of a new and important industry."
>
> - *John McCarthy*, MIT Centennial 1961 [Garfinkel and Abelson, 1999]

The creation of MIT's project MAC, a large research effort, had as one of its goals building a multiple access computer, this term being one of the sources for its initials. The outcome was

a research project known as MULTICS [Corbato and Vyssotsky, 1965] which, in cooperation with various commercial partners, intended to make possible the concept of the computer as a managed utility. Since computers were huge and expensive machines, needing special care and handling, they would be centrally managed while anyone who needed their services could 'plug in' to their network.

The theoretical contribution of MULTICS cannot be overstated. This was the largest assembly of many of the finest minds in the field, specifically considering the role and structure of an operating system. Their theoretical contributions to the field provided a host of innovations many of which continue on to the current day. However, like the initial time sharing system, its practical importance was supplanted by a quick hack[8] which contained a far more limited, but focused, interpretation of the MULTICS system. This program eventually becoming known as Unix[9] and remains the dominant research operating system in the world today.

The reasons for this are two fold. While the MIT history [Garfinkel and Abelson, 1999] is able to truthfully report that "Multics fulfilled virtually all of Corbato's goals" the users had less complimentary things to say. Doug Mc Ilroy states [Salus, 1994], "Three people could overload it". The 'hackers' [Levy, 1984] within the MIT labs were even harsher, "It was so slow that the hackers concluded the whole system must be brain damaged"[10]. The second reason is that the emergence of cheap and powerful computers, as exemplified by the PDP-7 on which Unix was first implemented, was beginning to call into question the utility model on which Multics was based.

It is also worth noting that both of these systems believed in a strong correlation between the language and operating system layers. MULTICS was going to be written in, and to work best with, the PL/1 language. The complexity of that language, and the resultant poor quality of the compiler, proved to have a strongly detrimental effect on development. Unix was written in C, a language as stripped down compared to PL/1 as Unix was to MULTICS. The result is that both systems could develop a very coherent interface between system and language.

DEC, the makers of the PDP-7, were less impressed by Unix. They had their own operating systems, which were also influenced by MULTICS, and a strong distaste for externally

[8] 1. n. Originally, a quick job that produces what is needed, but not well. (Jargon File).

[9] Unix is a pun both on how many features were lost and that the 'multiplex' aspect of utility computing was scaled down in favour of 'uniplex' usability.

[10] The source text also makes much of the MIT programmers hating the strict accounting that Multics, being intended as a public utility, enforced. They eventually wrote their own system, the Incompatible Time Sharing system (ITS), but it did not propagate.

developed systems. This is not too surprising, as traditionally operating systems had always been the vendors responsibility. Steve Johnson [Salus, 1994] recounts infuriating a prominent DEC executive when he summarised that their attempts to emulate the features of Unix, as a half-hearted compromise, "didn't work".

> "And I think Cutler's disdain has been reflected in his work ever since. Cutler was doing yet another OS based on a new architecture called Prism, not Unix, during Digital's internal RISC wars. Initially Cutler's OS wasn't portable, but was culturally compatible with VMS. There is a lot of stuff in NT that I think can be traced back to Prism. [Cutler went to work for Microsoft around 1983]."
>
> - *Steve Johnson*

Lastly there is another reason for the success of Unix, which echoes back to the very start of this document. Just as the lack of patents had allowed the ENIAC and EDVAC technology to have a widespread impact Unix itself gained a reprieve from commercialisation. Specifically Bell Labs, or more precisely its parent AT&T, was operating under a 'consent decree' (due to accusations of it being a monopoly) that disallowed diversification into new commercial ventures and required it to cheaply license any patents it held. The end result was that Unix became a communal resource to the nascent computing community.

2.5.2 Unified Resources

There was another significant operating system development that ran parallel to this effort, but with a very different focus. IBM was the largest supplier of computing hardware but it had many competitors, some of whom had better systems than IBM. They also realised that software was becoming a deciding factor in purchases. IBM wanted to innovate at the hardware level without invalidating the commercial advantage that an established software base gave it. In addition the cost to IBM of rewriting system software to accompany each hardware platform was growing at an alarming rate.

This problem, while it has a different origin and goal, is not dissimilar in effect to the previous concern. It is a recognition that while the compiler can translate from source to executable it cannot solve the problem of a changed environment. If the hardware environment, or the software

environment, changes there is a reasonable chance the compiler, or the binary it produces, will fail. The solution is the same as well: another layer of indirection to provide abstraction so that machines presented a standardised interface.

The project was OS/360, and a family of computers, which even for the giant corporation was a massive investment. Fortune magazine quoted one executive as saying, "we call this project, 'you bet your company' " [Shurkin, 1996]. The construction of the software, since it was intended to be a truly general system, was so complicated it spawned one of the seminal books on software engineering [Brooks, 1995]. It was also a commercial success, enshrining the advantage of a standardised system interface. In some ways it marked the dominance of application software over system considerations.

OS/360 did not have the same interest in interactive use that MULTICS did, being intended to continue IBM's dominance in business computing. This provided one of its least popular additions in the form of a Job Control Language (JCL). Effectively the functions the system provided were so varied that controlling it, in a non-interactive batch paradigm, was extremely complex[11]. This aspect faded as IBM adopted interactive use and time-sharing into later revisions of the system.

In passing it is worth mentioning that OS/360 was also intended to have a standard language, once again PL/1. However in the face of established applications, and the relative ease of adapting a compiler against re-writing all the applications, this attempt at standardisation was unsuccessful.

2.5.3 System Elements

The discussion of integrated systems has been in rough historical order. At the same time it has indicated three different and potentially opposed pressures that continue to exist in modern operating systems. The integrated system section focuses on the user view. As far as the user interacts directly with the operating system they want the system to assist them in "using" it towards their own goal. Many users may want a heavy focus on the interface and interactive performance. Application developers, a class of user who interacts closely with the operating system, want a fully featured API (application programming interface) that they can use to access the system's features.

[11] Although various sources have also emphasised that design flaws in JCL itself added to, rather than reduced, this complexity [Weinberg, 1971].

Section 2.5.1 on shared resources can be considered to be the view from the system perspective. All modern operating systems must be able to support parallel tasks (which may represent parallel users) all of which will be making demands on the available hardware resources. The system can know the mind of the user only through the API calls that are made on their behalf. These API calls are often designed for programmer convenience rather than that of the system. Since these calls provide limited context for the intent of the programmer the system designers will attempt to create strategies and mechanisms that optimise the expected usage. At the same time their approach must provide acceptable performance even for usage they do not expect to be common because it is still possible. This creates a challenging design problem.

Finally section 2.5.2 on unified resources indicates another view, that of the manufacturer. The hardware manufacturer wants the operating system to help amplify the advantages, and cover the weaknesses, of the hardware they wish to sell. It is advantageous if the API can be maintained, as much as possible, to avoid forcing users to discard valuable software and knowledge. The systems will be marketed for different uses, which will also change how they manage shared resources. The goal of taking advantage of unique hardware capabilities, without changing interface while allowing flexibility in policy forces more complexity into the operating system.

2.6 Commodity Systems

One of the most significant events in computing is the micro-processor revolution. It was yet another occasion which was ultimately shaped by a failure to secure the fundamental IP behind the system. Specifically IBM, which believed the micro-computer was either an over-powered terminal or a severely under-powered toy computer, designed an open platform with a third party operating system [Shurkin, 1996]. It succeeded in the immediate goal, providing an IBM badged competitor that dominated the fledgling micro-processor market, but IBM quickly lost control of the system they had so drastically underestimated.

For operating systems, the PC revolution was an uninteresting and regressive step. This is primarily because the process was hardware driven, harnessing the massive price and performance benefit that massively integrated circuits offered. It was also a different community from the one that had worked on Multics; this group was proud that they owned the computer and it existed purely to serve them. The following quote, in an article on virtual memory development,

gives a feel for this aspect:

> "You may have wondered why virtual memory, so popular in the operating systems of the 1960s and 1970s, was not present in the personal-computer operating systems of the 1980s. The pundits of the microcomputer revolution proclaimed bravely that personal computers would not succumb to the diseases of the large commercial operating systems; the personal computer would be simple, fast and cheap. Bill Gates, who once said that no user of a personal computer would ever need more than 640K of main memory brought out Microsoft DOS in 1982 without most of the common operating system functions, including virtual memory."
>
> - *Peter J. Denning* [Denning, 1997]

The first PC operating system was called CP/M [DR] written by Gary Kildall. Compared to the current state of the art in operating systems it was extremely primitive. It functioned more as a program loader, in the tradition of the old *monitor* systems, than an operating system. Indeed a running program could delete most of the operating system to get more memory, which was a scarce commodity, for itself. Interaction with all but the most basic hardware had to be addressed in application space. It did have the advantage of a modular construction, making it relatively easy to port to new hardware.

Tim Patterson, working for Seattle Computer Products, produced an unlicensed clone of a subset of CP/M called QDOS[12]. The company sold kits based on Intel's i8086 chip and the lack of an official version of CP/M was holding them back. A tiny company called Microsoft, that had been told by IBM to acquire an operating system in a hurry, bought it, fleshed it out, and sold it to support the IBM/PC when it launched. The IBM/PC platform became the standard microprocessor system, and Microsoft the dominant system software supplier for this new platform.

The important point is that operating system software was a requirement for having a marketable system, not a focus on which competing systems would be judged. As a result technical innovations, as opposed to popularisation, in the operating system domain was sparse. In time the system, now a commodity product, grew sophisticated enough to both support and require a

[12]Quick and Dirty Operating System

full-featured operating system. The dominant operating system on the modern PC is Microsoft Windows with the second being Unix (including the modern Apple OS/X). Both of these systems can trace their family lines back to MULTICS, Unix directly, and Windows via NT and Prism[13].

This period in time, the start of the PC revolution, is a good point at which to conclude this history of operating systems. The reason is partly because the massive growth in computer usage led to events and innovations being more widely noted. The other is because the systems environment had reached a position of stability. The commercial operating systems continued to evolve but they did so within the foundations that had already been established. While there are an immense range of variants and innovations (the family tree of Unix derivatives is a complex history in its own right) new commercial and experimental systems have not come close to challenging the dominance of the established systems.

One possibility is that many large segments of the market are not interested in the operating system per se. The general computer user will interact primarily at the application level and is interested in operating systems only as a required foundation for these applications. A dominant operating system will also tend to have a richer application set which increases its appeal. The next chapter will consider whether there are other, smaller and more specialised computing domains which place more emphasis and more stringent demands on the capabilities of the system.

[13]The true technical heritage of Windows is difficult to ascertain since it is a proprietary technology with a single vendor. This means there is no 'paper trail' accessible to the public.

Chapter 3

System Domains

The previous chapter presented a history of computing system development to a point at which they were reasonably mature and widely used. In the course of doing so the number of individual systems and innovations mentioned was fairly low considering the length of time computing systems have existed. The reason is not that computing systems are the products of a unified theoretical model that allows only one way to do things. There are a large number of operating systems that have been constructed. The reason is that the field tends to be dominated by a very small number of systems holding a massive statistical dominance. These systems attract most of the attention, investment, application software as well as skilled users and developers. This forms a positive feedback cycle which tends to shadow smaller systems, and starve them in terms of commercial possibilities.

This does not invalidate operating system theory as dominant systems are still likely to have technical limitations and room for optimisation. Creating models that help understand the internal structure of an operating system can help locate and analyse these opportunities for improvement. The model and the resulting analysis allow for the construction of experimental systems which will not attempt to compete commercially but can demonstrate that the idea has merit and could be integrated into other systems. These systems will tend to focus on mechanism and reuse the structure of the system being analysed. This is effort efficient and allows the changes to be more easily integrated back into the system that motivated the investigation. This allows system implementers to focus on the parts that incorporate their innovation without having to redesign all the surrounding system components which are required to construct a complete system. This approach does not work as well when the changes involve redesigning

the model around which the system is built. Fewer parts can be reused in construction and new concepts and innovations can be more difficult to integrate into systems that do not share the same architectural model. The same logic encourages system theory to focus on major system models and developments within the structure of existing system architectures.

There are also a number of specialised domains which have the capacity to drive operating system development. These domains will tend to be focused on the environment to which the system will be applied. This environment can vary from small embedded processors to very high powered computation environments such as avionics. The unifying element is that they are more demanding and focused than the more widely deployed general systems which must span all possible use cases. This gives the possibility of a specialised operating system having a strong local advantage over a general system within the domain it has been designed around.

There is a potentially strong synergy between operating system theory and such specialised systems. Challenging environments will provide the motivation to reconsider, modify or reinvent existing systems for a better result. Some facilities that are expected to be present in modern operating systems may be drastically simplified or absent if they distract from the overall goal of the system. Capitalising on these advantages will require the ability to model the system so that the potential advantage can be demonstrated and estimated in order to estimate the value of the project. The more efficiently systems can be constructed and analysed to prove the advantage of the end product, the more predictable the development process will be. The speed and reliability with which an efficient system can be constructed is important to making the process economically viable.

This chapter will examine some of the domains for which operating systems are constructed and the structural model the systems provide. This will include abstract systems where the model rather than the implementation is the focus. The motivation is to consider if these approaches to system design can be used more widely as a foundation for the design of novel customised systems.

3.1 Theoretical Models

Most complex fields have certain simplifying abstractions which make it easier to introduce a neophyte to the field. These demonstration systems work best if they are relatively complete expressions of system functionality and are accepted as an accurate working model by specialists

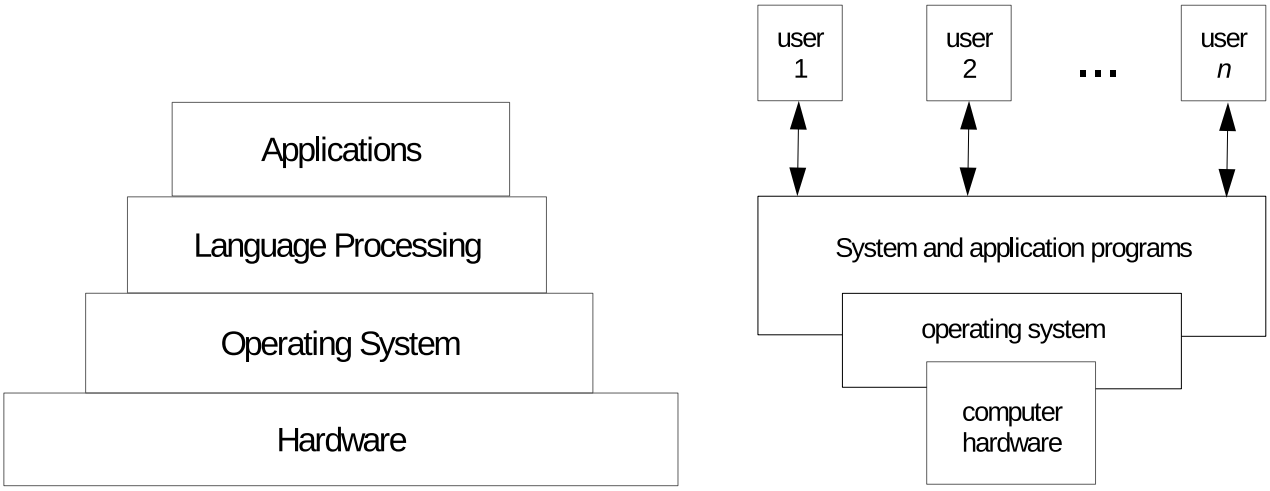

Figure 3.1: System Models

in the field. In that they are accurate models of a system, and take work to generate and explain, it is possible to think of these models as being representative micro-systems in their own right. In the operating systems field the gap in detail between theoretical model systems and deployed complete systems is vast.

The first development effort to truly focus on operating systems as primary entities, rather than simply a response to system issues, was the Multics project [Corbato and Vyssotsky, 1965; Corbato et al., 1972; Saltzer and MAC, 1974] which was covered in the preceding chapter. Generating a theoretical model of this system was not a design goal because of the perceived application for the finished system. Specifically it was to be modelled after the phone system, a utility providing services to all users with computing demands. As such there would be no real need for other operating systems since only large providers would be able to support the size of computer needed. This economic model also meant the users of the system had no reason to be interested in the details of the system as they would never deal with it directly.

Theoretical models only began to arise once the educational problem of communicating the principles of operating systems became prominent. One early text [Lorin and Deitel, 1981] provides an example of a model that will still be familiar. The important parts of this diagram

are reproduced in figure 3.1 on the left. In this model there are four levels of a computing system. The lowest and widest level is hardware which acts as a support for the higher and narrower levels which are, in order, operating system, language processing and applications. While the operating system is recognised as an independent component of systems, there is no depiction of it having an internal structure. The text instead provides a list of technical issues the operating system must deal with. It is defined by the services it must provide rather than what an operating system is. Indeed the authors discuss their inability to provide a high level definition.

> "We have a fundamental problem in defining what an operating system is and what it does. There are different opinions about what functions are applications functions and what functions are system functions to be implicitly provided by the extended apparent architectural view provided by the operating system. We have various points of view about what components of a software package are part of the operating system and what are separately package-able products... In general an operating system is a vague concept based upon tradition." [Lorin and Deitel, 1981]

A much later text [Stallings, 2001] is still using the same model although language processing is now considered amongst the more generic utilities. It is worth noting that this book, and indeed several of the other operating systems books still in wide use, have quite high version counts. This says a great deal about the lack of advance in the field, or at least the teaching of it. The text also references an older article [Denning and Brown, 1984] that presents an alternative hierarchical model. It offers 13 levels of functionality ranging from electronic circuits at number one to the shell at number 13. The increasing number of levels does little to aid clarity however.

A more modern text [Silberschatz et al., 2002] offers another version of effectively the same four layer hierarchy. This same model is also used in [Silberschatz et al., 2013] and is represented in figure 3.1 on the right. The changing importance of elements can be observed in the graphical depiction. In this depiction hardware is the lowest and narrowest category reflecting that the attention has moved away from a focus on hardware issues. The layers overlap representing that the dividing line between the levels is not precise. This could be seen as indicating deep interactions between the layers and a difficulty in determining a precise line of division. Language processing has now become simply another application rather than a system layer in its own right. In comparison the multiple users of the system are depicted as being clearly separated

entities having independent interactions with the top level. As such the diagram shows the increasing importance of application software which now overlaps the system and the growth of multi-user systems. The operating system still remains a closed and unexplored component. Once again, the authors express the difficulties in providing a firm model of an operating system.

> "In general, however, we have no completely adequate definition of an operating system. Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system... In addition we have no universally accepted definition of what is part of an operating system." [Silberschatz et al., 2002]

One of the most complex models of an operating system [Tanenbaum, 2001] combines several of the previous models. It begins with a system model which is the familiar hierarchical layer model. However this one is interesting in that the hardware layer becomes three layers (machine language, microprogramming and physical devices) which is an unusual distribution. Microprogramming is essentially a sub-component of hardware, while machine language is either an interface or a language. The operating system is also grouped with system utilities, including the compiler and shell, to form the category of system programs.

For the internal structure of an operating system four quite separate architectural models are given. This acts as a potent argument against the existence of a single theoretical system model. The structures given include the monolithic systems, which are considered to have no structure. Layered systems are introduced with the THE system [Dijkstra, 1968] and Multics [Saltzer and MAC, 1974] being mentioned as systems that exhibit this structure. Virtual machines, which are hardware multiplexers that can run multiple operating systems simultaneously are next. In this environment the operating system is effectively being run as a client, even though it has clients of its own. This leads neatly onto the final model, the client-server architecture, which includes distributed and microkernel based systems. These systems, which depend upon an ideally minimal infrastructure to connect system services run as clients, are somewhat like a specialised form of the virtual machine model. It is worth noting that virtual machine and layered operating system models are specialised, and even the client server operating system model will be unfamiliar to the general computer user.

In short there does not appear to be an agreed upon, and widely distributed model of an operating system. In such an environment individuals seeking to discuss operating systems are likely to find that there is a lack of common context which adds confusion to any communication.

The prevalence of the simple layered hierarchical model, for such a long duration, could seem to be a matter of some concern for the teaching of operating systems.

3.2 Educational Models

> "operating system design is mostly black art and little science" [Comer and Fossum, 1988]

There exist some systems that attempt to straddle the gap between theoretical models and production systems. The idea is that by showing a practical model of an operating system the application of theory will be observed and better understood. And since the system is not bound by commercial or application pressures it is able to remain relatively simple and cleanly architectured in order to aid comprehension. These systems find a viable market in those studying operating system principles.

John Lion's book [Lions, 1996] is one of the most celebrated of these. Originally constructed as class notes for a course, the content ended up having a wide distribution and a significant impact. They are directly derived from the then current Unix kernel, version 6, for which source was available. This is part of the reason for its fame as the source was not intended for distribution outside of authorised license holders. However the clarity of the source, then less than 9,000 lines of code, supported by commentary from a skilled practitioner, gave it a much wider audience. The code is targeted at a PDP11, a popular machine of the time and from the family of computer at which Unix was originally targeted but one unfamiliar to the majority of modern readers.

Another example is the Xinu [Comer and Fossum, 1988] system which moves closer to being a practical system. It was influenced by the THE system [Dijkstra, 1968] which was an attempt to impose a regular and clean architectural structure into operating system design. It can be seen as one of the foundations of the layered model for systems that is now taken for granted. It also seemed to offer a modular view of the system in which each layer could be reasonably simple but form a complex system in combination. The Xinu system consists of 8,000 lines of code and was most successful as an educational system.

The final example is a system constructed entirely for use as an educational tool. The Minix system [Tanenbaum, 1999] was coupled with one of the clearest books on system architecture.

This text was later extended and decoupled from Minix to form another text [Tanenbaum, 2001] for those not interested in implementation concerns. The Minix operating system was inspired by version 7 of the Unix kernel, represented about 12,000 lines of code, and, like Xinu, ran on the popular (and cheap) 8086 computers. The operating system was practical, partly because if purchased it also came with a collection of tools and utilities, and developed a community that put it ahead of other educational systems.

This community was partly the result of the USENET news system which allowed a geographically diverse set of Minix fans to communicate freely. This community also provided an answer to one of the limitations of these educational systems in general. Volunteers in the community would develop the utilities and applications that were required to make a system actually useful. Likewise they would fix bugs and provide extensions, something a purely educational system would not have the resources to do.

However the goal of being an educational system and the communities interest in a practical system was not without some pressure. Operating systems were growing in functionality, but attempting to add those features to Minix would increase the complexity of the system and thus reduce its educational value. It would also mean the book that accompanied the source would be out of synchronisation with the actual code. For this reason the author disallowed extension of the core source, although experienced users were permitted to patch in advanced features. It was this environment that became the seedbed for the nascent Linux system.

It is possible to argue that Linux, by becoming popular and available to those interested in operating system internals, closed the market for educational systems. However it may be more accurate to say that these systems actually became less viable as the complexity of commercial operating systems grew, especially those without a Unix heritage. The model systems were no longer true representations of real systems. There was also increasing competition for space on academic curriculum from new domains in computer science, as well as a possible perception that operating system internals were not a rewarding course of study.

Modern operating system texts[Silberschatz et al., 2013] attempt to be abstract in their approach and not be based on any particular system. This includes any attempts to base the text around a simplified educational system. They are more likely to include extensive case studies and encourage experimentation on full fledged systems, such as Linux, to support the main text.

3.3 General Systems

The personal computer made computing power a commonly available resource. The explosion in modern portable forms such as laptops, tablets and smartphones means many people constantly carry a powerful computer around with them. The ambition towards wearable computing seeks to extend this trend. This is the result of commodification of computer hardware and means that access to computing power is more affordable than ever. The amount of processing power available would have shocked the first proponents of personal computing. Indeed it was IBM's belief that the platform would never mature, and thus was not worth making proprietary, that allowed the open PC architecture to become ubiquitous.

The desktop general systems environment is also an effective mono-culture for operating systems. A 2001 report [Johnston, 2001], quoting the research group IDC, attributes 92% of operating system sales to the Windows family of operating systems made by Microsoft. A later report [Rohde, 2003] estimates the number at 93.8%. A large web site like Wikipedia can also gather statistics on operating systems used to access the site. In 2012 these numbers reported growth in the usage of OSX (Apple's operating system) has increased but Windows still commands roughly 90% of the desktop market. This gives them not only an effective monopoly position but it also gives them a massive benefit in terms of public perception. There are many computer users who will think of operating system and Windows as being synonymous terms.

This is not a beneficial environment for operating systems research. Microsoft's operating system is entirely proprietary and its design process exists entirely within the company. With a single powerful vendor there is no interest in establishing standards for anything other than external interfaces. Even in that case, since Microsoft is such a dominant vendor, if it abandons the standard then that standard will cease to be viable. This makes their observation of standards, even of their own creation, entirely optional on their part. In short Microsoft is dominant but, from an operating system development point of view, its closed and commercial nature does not provide a good foundation for research projects. The proprietary and secured nature of many modern operating systems provides an impediment and disincentive to operating system research.

There has been a significant growth in powerful non-desktop computing platforms. Smart phones, tablets and web appliances have all begun to challenge the dominance of the desktop computer. These systems are almost exclusively sold with and secured to the purchased

hardware. This means that there is little interest in or ability to provide alternative operating systems on these platforms.

Another market report [Rohde, 2003] states that in 2002 the combined desktop Linux sales were 2.8% shared between multiple vendors [1] . For various reasons the true usage figure might well be higher, but this figure does represent the revenue available to fund commercial development. A major part of the reason for Linux's survival and growth is that it is extremely promising for developers, including operating system developers. It has source availability, freely available development tools, a permissive license and a community supportive of such efforts. Its architecture, being based on the well established Unix model, is familiar to many developers. On the negative side the success of Linux has made the establishment of other alternative operating systems extremely difficult. This includes HURD [Bushnell, 1994], which was intended to be the completion of the system tools Linux uses and Plan 9 [Pike et al., 1995] which was the intended successor to Unix on which Linux is based. Its effects on Minix [Tanenbaum, 1999] have already been mentioned.

The number of alternative systems increases as one moves away from the desktop. Portable and embedded systems have a massive installed base, however they tend to be invisible, capacity restricted and unwilling to invest in operating system research. The Itron project [Takada et al., 1998] represents the Linux system being brought into this domain which will become more common as even embedded systems grow in capacity. Many of its competitors are proprietary operating systems which discourage research in that domain.

Moving in the other direction the high end server market exhibits a wide diversity of operating systems. Unlike the embedded market there are comparatively few installations but the amount of money invested in each is substantial. This encourages multiple operating systems in order to maximise the infrastructure. These do tend to have a restricted development community however and are frequently vendor specific, the operating systems being seen as a marketing advantage for the hardware. An example is the IBM Z system servers [IBM:2002] which support multiple operating systems the average user will never have heard of, including TPF, Z/VM, Z/OS and OS/390. It also supports the Linux operating system, although as a virtualised application rather than as a core operating system.

[1] Any mention of operating system market share should be viewed with suspicion. It is a field prone to advocacy and proprietary analysis in which there are major concerns on what is actually being measured. The important point is that while Windows is strongly dominant other systems (including a resurgence for Apple Computers OS-X) still command viable markets.

There are also a substantial number of small research projects that focus on constructing a general purpose operating system. These seem to be primarily motivated by the desire of one person to demonstrate breadth of skill by completing a full operating system, even though the resulting systems tend to be so crude that they are not truly usable. These master-work kernels have strong echoes in the creation of Unix and Linux which are also considered to primarily originate from a single author. However, forced to compete with far more mature and entrenched general systems, and unable to prove specific advantage since they follow the same model, these projects generally do not establish a viable niche.

3.4 Hardware Dominant Systems

One of the primary drivers of operating system advance are developments in computing hardware. The launch of novel hardware environments represents a substantial investment of development funds. Since without software the competitive advantage of the platform will not be accessible these projects are also more likely to consider investment in the system domain. However this investment, which is effectively purely a supporting requirement rather than a goal, tends to focus on specific application and minimal implementations.

This has proven a problem for hardware advances aimed at the general computing market. In this domain the investment required in order to develop a competitive system is extremely large. At the same time it is hard to clearly demonstrate sufficient competitive advantage to mobilise the entrenched general systems to adapt themselves to new technology. This can be seen in cases like Intel's i2o project [I2O:1997] which sought to generate a new standard for intelligent I/O peripherals. Despite being well funded and supported by many large names in the industry uptake of the new technology was marginal and the project's distribution web site has since been allowed to lapse.

Development efforts with less resources have an even smaller chance of broad acceptance. The reason is that, from the point of view of the generalised systems, the environment they occupy becomes faster without any change on their part due to general hardware advances. Meanwhile many domains thought to need specialised operating system support, such as multimedia, have become part of the general domain as system performance grew. In broad terms increasing global performance has proven to be an acceptable substitute for local advantage. This situation may change if advances in processing power become harder to achieve [Thompson et al., 1998; Mann,

2000] but this is not currently seen as an immediate concern.

One advantage of solutions based on a general operating system is that there is a potentially much larger number of users to fund any updates required to operate on newer hardware. These systems are also less focused on exposing hardware features to the applications programmer favouring a more abstracted view of the hardware. This allows allows new hardware and updated operating systems to be added to an existing system boosting its performance. By comparison many hardware focused solutions are, logically, closely tied to a specialised hardware implementation. The advances they make in a specific environment inevitably exact a cost in their applicability to the general domain. Many of these research projects are not constructed using the very latest technology, for economic reasons, and the progress of hardware can quite quickly render them obsolete. For example the MIT Alewife machine [Agarwal et al., 1995] is an interesting experiment in hardware assisted distributed shared memory. However the implementation of this system was on 33Mhz modified Sparc processors. A 16 node Alewife system is quite possibly slower, even for well suited code, than the average domestic computer a couple of years after it was completed.

The operating system innovations contained in the Alewife system remain of interest. It is not unusual that operating systems advance on a much slower scale than hardware. However given that the hardware will determine the user population there will be an effect on the motivation for research, development and technology dissemination. Since the Alewife system is closely tied to the specifics of the hardware it cannot trivially be moved to more recent hardware. In short there is a substantial possibility that system advances will be lost simply due to the process of implementation obsolescence.

There are various other systems that have occupied hardware centric niches for some substantial period of time. Real time computing platforms like VxWorks[2] (as used in the Mars Rover projects [Wilcox et al., 1995]) and embedded computing [Forin et al., 2001; Stankovic, 2001; Takada et al., 1998] have proven challenging for assimilation into general systems. This is because they represent a specific property which is architecturally opposed to the model of general systems. They trade against the size, complexity and relatively slow response times that are inherent properties of a general system, although as hardware grows in processing power these advantages may also be challenged.

[2]Being a commercial and proprietary product technical papers on the structure of VxWorks are not readily available.

The first two embedded computing examples [Forin et al., 2001; Stankovic, 2001] are also interesting cases. The embedded computing market has long had an interest in component operating systems. Such an approach allows the efficient construction of specialised systems which is also the goal of the Shards system. The direction of interest differs somewhat in that embedded component systems primarily want low cost of construction and minimal systems but are not interested in architectural flexibility. Thus they tend to have a well defined core architecture into which appropriate primitives are placed. It is a logical consequence of the field being dominated by micro-design imperatives.

Finally there is also a tradition of small groups focused on writing a 'super fast' operating system that fully exploits hardware. This comes from a belief that a implementation focused bottom-up approach will avoid the performance limits of current operating systems. Since these attempts are not theory driven or executed within a university environment they do not tend to generate research papers. Some current examples, eternally incomplete, include V2OS and DexOS [V2OS, 2002; DexOS, 2011]. These systems tend to champion extremely small kernels frequently written in assembly language. The claims they make about high performance and small size are appealing but most researchers in the field recognise that the development of Unix was the final proof in the advantages of high level languages for system construction. These systems are also unlikely to be able to demonstrate sufficient economic advantage in order to compete against the entrenched general systems.

3.5 Software Dominant Systems

> "An important long-term goal of our work is to explore the design of new software systems from a 'language-centric' viewpoint." [Harper and Lee, 1994]

Programming languages underwent a schism fairly early in their development. One group, best typified by the C [Kernighan and Ritchie, 1988] programming language, believed in programming to the machine. Another group had the idea, perhaps first embodied as part of COBOL but more popularly remembered in Lisp, of programming to the thoughts of the user. Naturally enough the first had a low level bias, valued compiler technology and prized efficiency. The second group had a high level bias and preferred virtual machines and software constructs for their endless flexibility.

Operating system design has been dominated by the first group. However the second group, who believed the expressive power of their languages made up for their performance deficits, also considered operating systems a viable target. This was assisted by the fact that the Lisp programming language pushed the capacities of the hardware on which it ran. This lead to the development of the Lisp machine [Withington, 1991][3] which was a combined hardware and software environment for Lisp programming.

There was another form of synergy between these advanced languages and dedicated systems. Many of the *functional* languages that came after Lisp included what could be considered system operations within the language structure. This is partly because, being built on virtual machines (software interpreters), they carried their own operating system with them. However it was also because the language designers believed that their languages were capable of superior expression for those primitives. Thus there have been a steady stream of efforts to effectively merge the language with the system environments, the ML [Harper and Lee, 1994] language being one of the most popular targets.

These development efforts tend to include one of two goals. On the one hand they aim to gain advantage by rewriting traditional systems in the target language. On the other they intend to re-structure the operating system's interfaces and operations so they are optimal to the needs of the language at run-time. This means they may accept some generally sub-optimal operating system mechanisms if they help minimise a specific performance concern within the language implementation.

The primary weakness in these efforts is not technical. There are technical concerns because many of the languages do not naturally match the tightly coupled nature of operating systems or the unstructured demands of hardware. However these pale in comparison to the fact that the domain of programming languages is extremely segregated. An operating system optimised to one language is likely to be sub-optimal on many more. The outcome of enforcing the language as an interface is that it reduces the potential population of users for the system, a process magnified by the fact that the architectural languages, which are generally the motivator for language based operating system's, are not dominant.

There was an attempt to write a JavaOS when the language was new and growing rapidly in popularity. This would be an operating system perfectly suited to the operation of the

[3]The software component of the original Lisp machine remains a commercial product, albeit much reduced. See http://www.symbolics.com/

language, ideally written in Java itself, and tightly integrated with the newly announced Java optimised processors [Wayner, 1996]. This had the potential to unify hardware and operating system software, with research projects beginning in both these areas, around this new language. Unfortunately both of these projects proved unsuccessful, which to an extent cast doubt on the viability of the concept. The specific reasons for the projects failure would make an interesting investigation, although sadly much of the documentation for the project was sponsored and hosted by the owner of the language. Once the projects failed, information about them silently vanished which makes it hard to draw lessons from the work done. Some of the development frameworks [Back et al., 2000] can still be found, as can plans for the system itself [Madany, 1996] although detailed technical information probably never left proprietary status.

3.6 Design Dominant Systems

> "Virtually all operating system researchers realize that current operating systems are massive, inflexible, unreliable and loaded with bugs, certain ones more than others (*names withheld here to protect the guilty*). Consequently there is a lot of research on how to build flexible and dependable systems. Much of the research concerns microkernel systems"
>
> - Andrew Tanenbaum [Tanenbaum, 2001]

As indicated in the quote above recognition that the size and complexity of operating systems is impeding research and development is not a new occurrence. It is probably safer to say that every researcher who has worked within the field has wished for a simpler foundation on which to build their contribution. The microkernel architecture is a specific approach to the general goal of a well structured operating system. In short it is an operating system where the design is architecturally clean.

This has a very powerful attraction to operating system researchers, not only because it makes operating systems easier to work with but because it is an appealing idea in its own right. The concept that there is a simple and elegant core, that the perceived complexity is primarily accidental or unnecessary, represents an architectural goal in itself. However in practice, because this architectural discipline places constraints on run-time behaviour, there is a performance

penalty. Since architectural structure, regardless of elegance, is not directly visible to users there has been some resistance to accepting even a quite small performance penalty. This was one of the elements in the well known Tanenbaum / Torvalds e-mail argument [DiBona et al., 1999] that represented the two positions, the theoretician versus the pragmatist.

The microkernel approach remains the most active branch in operating system research. There have been an immense number of microkernels constructed. This is partly because, being minimal, they are more viable targets for a research project. One of the best known is Mach [Rashid et al., 1989] but one of the best regarded modern microkernels is the L4 [Haeberlen et al., 2001] kernel which has been refined through many variants and used as the foundation for many projects. The current version is being used as the base for the ambitious Sawmill [Gefflaut et al., 2000] project which aims to reduce Linux to a collection of modular servers.

This project, which has similarities to Shards in the desire for modularity, also represents one of the disadvantages of microkernels as a foundation. The microkernel design calls for system configuration to be expressed in the run-time interaction between servers (which are basically executing system modules). This is the source of the performance concerns with this architecture. The L4 system is regarded as being extremely efficient, however systems that build on top of it inherit its run time approach but can rarely match the efficiency, partly because their scope is bigger. As with all microkernels it is possible to say that they can be made so efficient because, in terms of the whole system, they do so little.

There have also been variations of the microkernel idea built from more modern modular software mechanisms. Specifically Object Oriented systems [Campbell and Tan, 1995] involves a language approach to modularity being extended to the systems level. Other modular systems such as CORBA (Common Object Request Broker Architecture) have also been used as a mechanism for the construction of operating systems [Kon et al., 2000].

There is a conflict within these approaches however. The higher level modularisation mechanism are designed in light of the growing size of application software. As such they are generally less concerned with efficiency[4] and more concerned with reducing complexity through modularisation. They value robustness over flexibility. These are desirable aspects for large software projects, but they do not necessarily translate well to the system level where performance is a

[4]This refers to the object oriented model as a theory, not to the speed of specific code. Practical object oriented languages allow the degree of strictness in the application of the design theory to be determined by the programmer.

concern and the structure dominated by micro-architectural issues.

3.7 Application Dominant Systems

In theory the field of application dominant systems should be the largest field for operating systems advance. The term refers to an environment where the operation of the application is so valuable and demanding that any system component that interacts negatively with it will represent a possibly significant loss of value to the owner. Stated differently there are some applications where being a little slow, or a little inaccurate, or a bit limited in capacity, are completely unacceptable. These systems can often have very specific needs and distinct use patterns which could be supported by a system built to their needs. The cost of system optimisation would be justified by the improvement in the performance of this economically valuable application. The specific needs discovered in such case would also ensure a steady stream of research imperatives and opportunities.

However in practice very few examples of this sort of specialisation were found. This is not an unexpected result. The argument has already been made that commercial pressures call into doubt the viability of customised operating systems, regardless of the value of the client application. It was still expected that research operating systems, which do not have the same commercial pressures, may have done some exploration in this area. In practice it seems that time, personnel and funding resources, while no doubt accounted for differently, still provide restrictions in the research environment.

The main proponent of application specific optimisations are the extensible operating systems [Small and Seltzer, 1996]. These systems intend to formalise customisation to the local environment so that the infrastructure needed can be provided in a general core system. They have some similarities to the design dominant systems discussed earlier in that some parts of the system are invariant (the core system) and interact with components which may be flexibly replaced. They differ primarily in where this line of interaction is drawn. In microkernel based systems the interaction between software servers, with the protected mode[5] core acting as a facilitator, allow for adaptation through replacing one of the servers in the interaction. In extensible systems the software interacts directly with the kernel and is able to inject some of

[5]protected mode refers to hardware enforced levels of trust, with the kernel generally being more trusted and having more capabilities than user mode software. There is some cost in the *context switch* required to change modes.

its own code into protected space to run in the same context as the kernel.

The advantage, and disadvantage, is the tight integration between the core system and the extension. Being able to provide a general mechanism for totally flexible extension, which can also be protected from damage by hostile or badly written application provided code, is quite challenging. Some of the performance advantage from the tight integration is sacrificed in responding to these concerns. It also becomes clear that the definition of what needs to be extensible can only be formulated against some imagined application demand. In short what must be extensible for some system should be core for others, which discourages the notion of a single extensible system for all environments.

The result of this can be seen in a system like Vino [Small and Seltzer, 1994]. This system is extensible, however its determination of how the extension facility will be structured is shaped by its observation of commonalities in the system demands of an application family. In particular it noted that databases have demands that can be productively incorporated into an extensible kernel, specifically their I/O patterns. However there is no reason to assume that this delineation of what is core, and what must be extensible, is applicable outside of this focus. Thus ultimately extensible systems still incorporate a design goal and are not truly universal. Examination of other extensible systems such as Spin [Bershad et al., 1995a] and the Cache kernel [Cheriton and Duda, 1994b] all reinforced the observation that each of them was based on particular expectations of application behaviour.

This same approach can also be seen with custom-built systems. The Scout system [Montz et al., 1995] focuses on optimising a property that is expected to be of general value to applications. In this particular case the operating system advances a communication abstraction, the path, and elevates this as the focus of the design. If building an operating system optimised for an application is not viable then targeting a common (but probably not universal) behaviour is the most logical response. Interestingly Scout also leverages the compiler, but only for performance optimisation, rather than as a means of increasing flexibility.

The end result was that the number of system developments focused on application support was fairly small. It seems reasonable to argue that the current limits in development flexibility play a large part in this absence. However it is also worth considering that to some degree it might be traditional. Operating system developers have become separated from language designers and application developers, which means that opportunities to find and exploit synergies simply do

not arise. Even where an operating system is co-developed with an application, as is the case with the cache kernel (which was developed as part of a wider project) such connections are downplayed in order to establish the systems project as an independent entity.

3.8 Summary

A number of viable domains for operating system research were found to exist. However in many cases these domains were not large or had well established and mature systems that already fully supplied the needs of the domain. This led to little pressure for rapid innovation and production of new operating systems.

The potential of specialised application software needs to drive operating system development seemed promising. It implicitly demands multiple and specific solutions while having the potential to make providing a solution economically rewarding. The limitation was the effort and risk of constructing operating systems. This required a very pressing need to consider developing a custom system and thus only a small sub-set of the potential possibilities would be available.

At the same time there are a number of theoretical and technical approaches to operating systems which have created mature and well respected operating system components. What did not exist was a way to package the knowledge and mechanisms generated by these projects so they could be easily reused in the production of customised operating systems for specific application needs. Reducing the risk and cost of operating system construction could allow more demand for operating system production which would provide more opportunities for experimentation and discovery to drive development in the theory and practice of operating systems.

Chapter 4

Customised Systems

"Operating systems have become extremely large programs. No one person can sit down at a PC and dash off a serious operating system in a few months. All current versions of UNIX exceed 1 million lines of code; Windows 2000 is 29 million lines of code. No one person can understand even 1 million lines of code, let alone 29 million lines of code. When you have a product that none of the designers can hope to fully understand, it should be no surprise when the results are often far from optimal."

- Andrew Tanenbaum [Tanenbaum, 2001]

4.1 Introduction

The previous chapter indicated that there is a broad division that can be drawn through the operating system domain. There are *general* systems which are stable platforms for the widest possible range of usage. This category is dominated by a small number of long lived, stable and fully featured platforms with a substantial number of applications created for the system (e.g. Unix and Windows). This domain prizes stability above innovation, especially anything that requires dramatic changes in the application programming interface which reduces the amount of software available. As a result innovation tends to be evolutionary, in place and backwards compatible.

The more interesting testing ground for new operating systems ideas is the domain of *custom* or *specialised* systems. In this domain the system as a whole has a specific goal and the operating system must assist in meeting this requirement. The focus on a specific goal allows for more

dramatic innovation and means the narrower range of application is an inherent property rather than a commercial weakness.

One of the questions asked will be whether a specialised system can actually provide enough benefit to justify the substantial effort and risk required for implementation. The safer alternative will be basing the system on a general operating system and accepting that it may not be optimal and is not as flexible. The main part of the decision will be determined by the details of the implementation target. The other element will be any tools, frameworks and approaches that reduce the risk and effort of implementing a new system. These resources will be the topic of this chapter, leading towards a suggested approach for the efficient creation of specialised operating systems.

4.2 The Role of Pure Theory

The development of new operating systems or system software is, clearly, a small field in relation to the much wider role of operating system use and application programming. Since it is a small and specialised field the literature about the methodology of operating system design is extremely sparse. While there are a reasonable number of works explaining a design [Bach, 1987; McKusick et al., 1996; Tanenbaum, 1999; Raymond, 2004] that already exists there are few guidelines for those who are considering the creation of a novel system.

However the rarity of source material seems symptomatic of something more. After all, there are fields in which a vast amount of theoretical analysis is constructed on top of a relatively narrow range of practical application. But in computing systems there is no shortage of examples of practical application. Indeed deployed operating systems are so prevalent that they are taken for granted. These operating systems are usually considered mature and commodity foundations rather than project features in their own right. As a field of study the specific, in the form of a working system, dominates the abstract, the theories explaining why the system works in the fashion it does. Stated directly the issues within operating systems are often assumed to have been solved because implementations are so well established. This matches well with the vision of systems as engineered, evolved, stable and long lived development processes. It also matches quotes made by observers of the field:

> "The Unix philosophy (like successful folk traditions in other engineering disciplines)

> is bottom-up, not top-down. It is pragmatic and grounded in experience. It is not to be found in official methods and standards, but rather in the implicit half-reflexive knowledge, the expertise that the Unix culture transmits."
>
> - *Eric S. Raymond* [Raymond, 2004]

> "Talk is cheap. Show me the code."
>
> - *Linus Torvalds*

> "After 20 years, this [commentary on Unix V.6 source code] is still the best exposition of the workings of a 'real' operating system"
>
> - *Ken Thompson* [Lions, 1996]

In such an environment what value, or alternatively need, is there for general theories of system construction? Certainly the existing examples such as Unix, with their mature code, proven performance and wide applicability to application needs, prove that systems can be constructed in this way. The disadvantage is that the lessons learnt can not be expressed in a way that can be applied outside of their originating environment. If you are not a member of the culture expressed in the first quote, or are considering a system which cannot be expressed in a Unix compatible structure, then you are effectively forced to start anew. There are precious few system projects of sufficient size that they can consider construction on that scale.

Theory is also important in the judgement of alternative possibilities. The cultural rules and the idea of proof by implementation cannot extend to a system in planning if that system is substantially different. In simple terms, "this is how we do things" is not reusable or relative as different projects will naturally do things differently. The quote "this is the best way to do things" encourages reuse or productive comparisons with different approaches and the strengths and weaknesses they bring. This question also naturally leads to consideration of the focus of the system and the environment in which it will run which is rarely considered for general systems. The primary difference is that one is in a local context and the other one is in a universal context, even though this means it must carry much more detail within the description because it is unable to assume a common foundation.

The best example of this limitation is to ask whether the systems we have now are optimal, which is effectively asking for a universal solution to operating system issues. Or alternatively we can describe a task and ask which of the existing systems provides the best match to its needs, which is effectively asking for a locally optimal system. Computing systems as a cultural practice can not answer this question. In a similar fashion the ability to detect poor results, systems that are functioning as impediments to new possibilities, is hampered.

The final argument is that development communities, being informal and organic, routinely dissolve. The Unix community is fortunate in having remained viable for an extended amount of time, which is in no small part a tribute to the strengths of its design. However many other communities, quite possibly with observations and innovations that held competitive advantages, have not been as fortunate. In these cases, once the community disperses, the technical advances understood within that population can be lost. Documentation may remain, but it is frequently usage or specification oriented and misses the design logic behind the implementation details. This concern is magnified since development is frequently constrained by the privacy required for commercial advantage. However it has been a concern even in far more public systems:

> "While I felt that the rest of the committee was leaning towards recommending termination, I began to realize that killing off Multics was a terrible idea. My reasoning was as follows: While the Multics project might have been overly ambitious, it was the sole embodiment of a great number of very important ideas. The current efforts were taking advantage of lessons learned; knowledge and experience available nowhere else on the planet. If we killed off Multics, it was likely that these ideas would become lost art and would be discredited along with the whole Multics project; all to the possible detriment of many future projects. I decided that the committee had to come to the opposite conclusion to the one that [Prof. Licklider] expected."
>
> - *Ed Fredkin*[1]

Research in the systems field is often strongly bound to a particular project. The size and scope of most systems means there must be considerable economic incentive to finance a sufficiently large team for the period of time needed. The project provides the environment, culture,

[1]Taken from http://www.multicians.org/history.html

minds and challenges that encourage investigation and discovery. The novel aspects of the system motivate existing general mechanisms to be reconsidered for their relative applicability and encourage experimentation. The success of the system allows the results of these advances to be measured and promoted as having shown value in a successful system. If the project is ultimately unsuccessful or short lived then system innovations will often fade with the project.

This can be considered a weakness in the system design process. The first element of this is that the limited ability to reuse existing frameworks and components in construction adds to time pressure. This reduces the number of system projects that are viable and the time available for research over using something that "keeps it simple" and is known to work. In addition the focus is entirely on integrating any innovations. They are less likely to be documented or packaged as useful artefacts in their own right unless they are extremely concise, novel and marketable (such as a new algorithm). Novel mechanisms are seamlessly integrated into the product and as such largely unrecoverable without substantial reverse engineering.

The existence of a system development process would be important for the individual or academic researcher. They cannot rely on having access to a convenient existing system development effort or the funds to start their own. Such a process would mean that core mechanisms could exist independently of a system development effort. They would be products in their own right, would have value if created from existing systems and would be useful materials for future development efforts. They could also be scaled to the level of individuals and smaller teams who could focus on the production of reusable mechanism. The invention of novel fundamental algorithms would retain its value but the scope would become wider. How mechanisms could be packaged for reuse, integrated into future systems or how they could be specialised would become viable projects even if they were not aiming at production of a full system or new algorithms[2]. The study and improvement of the system development process itself would also be an area of research.

4.3 Initial Development

> "For when new ground is broken, it is usually impossible to deduce the consequent system behaviour except by experimental operation. Simulation is not particularly

[2]One of the assumptions is that Linux provides a mature and accessible general system and thus would continue to be a worthy subject in its own right

> effective when the system concepts and user behaviour are new. Unfortunately, one does not understand the system well enough to simplify it correctly and thereby obtain a manageable model which requires less effort to implement than the system itself." [Corbato et al., 1972]

This thesis focuses on the design and construction process of an operating system or system software project, rather than the end product of that process. If mechanisms can be packaged as artefacts independent of a particular system then there is potential for reuse. These mechanisms would have to be easily integrated and extremely flexible and configurable so that their structure did not limit the possibilities of systems being constructed using them. The Shards process will investigate how this approach could be enabled although it is itself a foundation for future development rather than a complete and mature system.

The development of Shards was to run in parallel with an operating system development project by a research group (Software Engineering Research Centre, SERC) within RMIT. This project was named Magnus[3]. One initial observation was the way in which the project was based entirely around a specific environmental goal for its commercial viability. The project had access to a very specific technical innovation in the form of a non-blocking, multi-channel optical switch. It was believed that this switch would have unique advantages if used to connect multiple computers as part of a powerful parallel processing system. The systems commercial viability would depend on how well the central switch could be supported by the system software and made available to applications that required those capabilities.

The foundation of the system would be this switch but the value would be in the operating system. The switch could easily be used by purchasing off the shelf hardware with Linux installed and considering the switch to simply be an unusually fast and expensive version of a standard switch. The extent to which the planned new operating system could recognise and support the unique aspects of the switch, and make them conveniently accessible to the applications layer, would be fundamental to the value of the system generation effort. Any locking, blocking or latency at the system level could rapidly degrade the advantage the new system would deliver.

The application layer would also become part of the solution. As a customised system the application layer could be constrained so that it would integrate with system design choices.

[3]The Magnus system was ultimately unsuccessful. Some of the technical details for the project are included in the introduction to the EC compiler [Castro, 2001], which was to be a development tool for the system. This is the only publicly available document with such information.

For example the system would only support one programming language which was well suited for parallel application development. The applications of interest would also be those that had high commercial value and could make use of the underlying system resources if they were made available. The system would not support general desktop applications at all as these would gain no advantage and could be run on standard desktops. This approach of building the system around its economic advantage is the definition of a customised system.

The development effort was also under significant time pressure which is expected to be the usual case. The commercial advantage relied on the novelty of the solution and resources were stretched by even an optimistic estimate of the development task. An estimate which had a very large degree of uncertainty since the system would be developed from scratch. The project plan simply did not allow for time to be invested on anything that did not directly progress development. This pressure was naturally concentrated on the lead designer who was fully occupied with absorbing the context in which he would be working. Attempting to generate reusable insights, for the advantage of others outside of the project, represented wasted resources and an unwanted distraction.

With no established process or intent to reuse components the issue of software development effort estimation was a key concern for the project. As indicated in chapter one the viability of a development project is determined by a direct comparison of risk versus reward, as is true for any large software project. The demands contained within the system design were specific, would require an integrated system and were intolerant of latency. The system software was likely to be the primary performance constraint[4] and the distributed nature of the architecture meant any penalty would be multiplied by the number of nodes (participating systems). In short the goal was extremely challenging and consequently carried a significant risk of failure. However estimating the severity of risk, and the potential for unforeseen negative interactions, was extremely difficult and magnified by each uncertainty in the projected outcome.

The reward, the gain for constructing this system over adapting an existing one, was extremely hard to judge. The system would be starting design from scratch so there was no baseline model or measurements to work from. In addition the specific application needs and priorities remained fuzzy as there was no clear vision of what application would best profit from

[4]The primary hardware limitation was estimated to be the computer's bus speed, rather than CPU or the network. Software will generally need many bus accesses to perform its function, especially in a SMP environment, meaning its behaviour would significantly impact on this constraint. This is part of the reason why cache behaviour, considered later in this thesis, was so important.

the systems presumed capabilities. Nor was it easy to visualise such software when the system itself was still undefined. This tight integration and resulting inter-dependence is part of what makes estimation so limited in the system domain. The result is that the relative benefit of constructing the new system was uncertain, while the risks were evident, which required the software equivalent of a leap of faith for the project to continue. However once initiated this investment actually served to discourage further effort in refining the accuracy of the estimation.

While one case is not statistically meaningful it seems quite reasonable to assume that many other development efforts face similar pressures. The system, as indicated in the Multics quote at the start of this section, is extremely hard to model. This is partly a result of its size and complexity but it is also amplified by time pressure and that system components tend to be tightly coupled. The demands upon the system will be determined by applications that do not exist at the time it is designed and whose design will depend on what capabilities and mechanisms the system provides. The paradox is that the application programming, which acts to test the system design, requires the system design as a target to be written for. In short it is this high level of inter-dependency that makes systems so hard to visualise and estimate. The lack of established theories, models and even in some cases basic documentation and reusable code provide a weak foundation on which to base expected structure and behaviour.

The practical solution to the complexity of planning, and the resource constraints, is to have a single person work on the high level design. This provides the design with some degree of internal consistency though it introduces a factor that has limited scaling. The larger the size of the project the more high level and abstract the core design will be as complexity and this scaling factor becomes an issue. More of the fine detail will be determined later by trusted designers or team leaders. These designers while they may test the details in conversation with others, become responsible for envisioning the complete architecture on which the system will be based. The designers are also able to deal with imperfect information or fuzzy requirements by simply making a judgement call based on their experience. The answer may not prove to be ideal but it is fast and avoids the design process blocking in attempt to provide conclusive answers while the process is still in flux and there are many unknown elements. This model is traditional in operating systems with Multics being seen as an example of failure by (and due to) design by committee, while the elegant and enduring Unix originated in a single mind. The counter-point that Unix could not have been written without the foundation provided by

Multics is generally not emphasised.

One disadvantage of this approach is that the logic behind the design will not exist outside of the head of the lead architect. The lead architect, being critical, has better things to do than write massive and complete documentation of his idea even assuming there are no intuitive aspects to their design. There will be personal discussion with other designers, so that they can work on their parts and understand the global system, and specific documentation for elements that operate outside the design group (such as the programming interface), but that will be all. Seen this way the fact that the main documentation released during Unix development was a manual for programming the system [Thompson and Ritchie, 1979] makes perfect sense.

The situations described above, seen in a wider view, can be identified as a self-perpetuating cycle. Because there are so few general theories and practices the systems are constructed in a quite informal manner, one individual relying on their personal knowledge and creativity to provide a seed solution. However because the solution is so deeply connected with their knowledge and problem solving methods no complete external representation of their design logic will exist. As a result of this lack there are no materials sufficiently complete to be the foundation for future projects other than having the core developer physically present to provide insight and advice to specific queries. Analysis and comparison are likewise impaired by the lack of insight into why particular choices were made.

The possibility of a more systematic approach, if possible and even if not complete, would offer two positive outcomes. The first is that its use would enable some elements of the design task to be simplified. The second is that the design, being shaped by the theory inherent in the inherited material, would also be easier to explain using the same terminology. In simple terms, material being reused need not be designed or explained. Material that is custom can be explained in comparison to what it has replaced. Even better, this raises the possibility of a positive feedback loop. The methodology, which is both strengthened and extended by use, becomes an even stronger foundation for the next system development effort. The result ideally is both iterative and evolutionary.

4.4 The Idea of a Universal System

A natural approach to the idea of system development is to imagine a single universal system. This concept can be considered as the ultimate technical challenge in the domain and would

effectively solve the question of system design if it could be completed. As such, research teams and individuals have been drawn to attempting to provide a universal system that could be adapted to any challenge. These attempts generally end in discovering that designing a system that is optimal in all environments has almost unlimited complexity and unresolvable trade-offs. This thesis started on the same path and made the same discoveries.

The core of the idea is to construct a universal, but task-neutral, operating system. The system, not being built for any particular environment, can be designed purely on the basis of the best theories and models for how operating systems can be constructed. In other words it would be a concrete expression of the best mechanisms for a general operating system. As such if a novel operating system being considered did not specifically need special behaviours in some area then construction could be simplified by directly reusing the universal code or section. Thus a universal core system acts as a foundation for the construction of systems.

If the design or implementation indicated that the universal system was not optimal for a new system new code would be required to provide a specialised variation. Ideally the custom operating system would determine which module in the universal system was incompatible and remove it. A customised equivalent would then be written to stand in its place. The new code would now represent either an evolutionary enhancement of the universal (if it was superior but could also function as part of the universal model) or as an alternative module that exhibited a fundamental design decision. Stated another way the first case would involve the new code being better in all usage (evolutionary refinement) or better in some cases (a local design decision). In either case the universal system along with its variant forms would be made more capable and thus better able to support future development projects.

Attempts to design a universal system revealed that the concept did not match particularly well to practice. The primary difficulty was that finding truly neutral and universal mechanisms was hard. In practice there are many competing pressures in an operating system, especially where resource management is involved. Without a specific environment, and goal, to provide design direction there was no way to craft a solution. Effectively there were too many possible answers.

Any given answer to a design concern impacted the system being constructed at many points. Since operating system components, the interactions between them and other components, were built in relation to a design goal they became conceptually linked. The result is that assumptions

began to bleed across module boundaries. This cross linking, and environmental sensitivity, make it clear that the basic idea of a universal system, and its ability to provide clear isolation of component parts, is not actually practical.

A more precise and immediate failing was that there was no reason for this proposed universal system to not look like Unix. Unix does not have a pure modular construction but as was indicated it is very hard and possibly impractical to avoid design assumptions spanning component boundaries. Proving that a system is truly universal, the best expression of operating architecture and mechanisms, is extremely hard especially when the design goal is so broad.

On the other hand an existing general system like Unix has the potent advantage of software maturity and proven applicability. Unix has a respected design as a general system, and has proven widely applicable such that is is a reasonable basis for a foundation design. It quickly became clear that the any attempt to construct a universal system, having no environment in which to prove superiority over Unix, will itself be displaced by Unix.

In other words, without a specific example of something Unix can not do well, or be made to do well with reasonable effort, there is no reason not to treat it as a de-facto universal system. In practice this removes the motivation to create an ideal universal system when a practical universal system is already available

4.5 The Idea of a Minimal System

Another approach is to focus on some of the sources of complexity, specifically the inability to design, test and defend a system without a guiding context. There is also the fact that the high degree of coupling in an operating system works against being able to treat it as a collection of modules. The natural alternative is to instead consider building from a minimal core. Such a system environment, since it spans only the most essential and primitive functionality, is less likely to limit or be affected by the design requirements for any specific system. Since the system functionality that will be modified, exists outside of this core, there is the possibility of strongly reducing the range of interaction between the two.

This approach is not novel, indeed it is probably the most common response to the design complexity inherent in operating systems. The resulting field invented what are termed microkernels (which were also covered in section 3.6), the most famous probably being Mach [Rashid et al., 1989] but a more modern version being L4 [Haeberlen et al., 2001]. While the designers

espouse a great number of goals for their systems it can be seen that the primary and unifying design goal is the control of complexity. This being achieved by having a core, that is carefully architectured and relatively static, while the variable elements of the system are contained in external microsystems.

> "The system becomes more flexible and extensible. It can be more easily and effectively adapted to new hardware or new applications. Only selected servers need to be modified or added to the system. In particular, the impact of such modifications can be restricted to a subset of the system, so all other processes are not affected. Furthermore modifications do not require building a new kernel; they can be made and tested online." [Liedtke, 1996]

The interesting observation [Rashid et al., 1989; Bushnell, 1994; Small and Seltzer, 1994; Bershad et al., 1995b; Liedtke, 1995b; 1996; Small and Seltzer, 1996; Haeberlen et al., 2001], is that this approach does not automatically reduce inter-dependency between modules. What it primarily serves to do is formalise, and narrow, the interface between the two. Rather than modules being able to directly call required functions they must work through a centralised control mechanism. However this mechanism, being the sole channel of communication and intentionally minimal, also acts to structure what interactions are possible. In short the details of how the interface is constructed will have a significant result in how higher levels of the system can be expressed, and those modules will be dependent upon the expected operation of the core. It can even be argued that as the degree of independence between modules grows the complexity of the interface, and the module's dependence upon the specifics of its behaviour, will grow in turn.

In a similar way there is no automatic reduction in the dependencies between modules. The reasons why modules might want to interact still exist after all. The difference is that they are routed through the microkernel rather than sent directly. As for the kernel-module case the interaction has been formalised, and the channel narrowed, but it has not been solved in a way that will remove the need that generated the interaction in the first place. It can even be argued that the addition of an intermediary step makes the interactions harder to recognise, and thus even more prone to being disturbed when extending a module.

The central limitation with the microkernel approach is much simpler: formalising, narrowing and restricting the communication within the system has been found to exact an execution

penalty at run time. The mechanism that allows separation, because it requires additional handling to maintain, cannot be as efficient as a direct connection. Even more seriously it has been found that many of the close interactions within the kernel exist for very valid and immediate performance reasons. The more modularised the kernel is, the more restricted and guarded the interactions, the more performance is lost. Stated another way if there was no gain from tightly coupled code in operating systems then Unix probably would be a perfectly modular system. However the fact it is not modular cannot be taken as evidence of error, but may in fact be a carefully balanced concession to performance.

> "Although much effort has been invested in μ-kernel construction, the approach is not (yet) generally accepted. This is due to the fact that most existing μ-kernels do not perform sufficiently well. Lack of efficiency also heavily restricts flexibility, since important mechanisms and principles cannot be used in practice due to poor performance. In some cases, the μ-kernel interface has been weakened and special servers have been re-integrated into the kernel to regain efficiency."
>
> - *Jochen Liedtke* [Liedtke, 1995b].

This remains largely true today. Even for the most advanced microkernels, such as the well regarded L4, it requires significant work to contain the inherent limitations of this approach. The more regimented the architecture the less freedom it has in responding to optimisation possibilities. Since inefficiency in the operating system applies to all uses and users of the system, while architectural purity benefits primarily system designers, it is reasonable to assume the cost versus benefits calculation will not be of interest to the general system user. This is amplified by the fact that applications are constantly and dynamically interacting with the operating system, if there is a performance penalty it may be paid many thousands of times per second.

> "The motivation that led to the emergence of microkernels in the early 80's leads now to the emergence of extensible operating systems. It is unreasonable to expect any one system to possess all of the functions needed by all applications. Rather, vendors of sophisticated applications, which require application specific operating system customisation, sell these extensions as well" [Small and Seltzer, 1996]

One answer to this problem is to attempt to keep the modularity in the design but allow it to be bypassed in the implementation. An example of this is the field of extensible systems which allow system code to be, possibly dynamically, placed within the kernel. This obviates the need for an abstraction mechanism and allows the code to call kernel components directly. The end result however is that the modular design and component independence that was the goal is being compromised. If the implementation is not going to be cleanly modular then the value of the approach is called into question. It might be more correct to have a design that recognises that there will not be clear lines of division between functional elements and includes that explicitly. There is also the highly problematic issue of security when code, which might be end-user supplied, is effectively able to operate at the same level of trust as the system kernel.

4.6 The Idea of Modular Assembly

The concepts behind extensible operating systems offered the greatest potential as a framework for creating customised operating systems. They did not provide a solution but they allowed customised code to be deeply integrated into the implementation. The concern was that the implementations tended to be complex. The security and integration mechanisms required a considerable amount of run-time cleverness to work. Some of the approaches, such as containing extension code within a virtual machine, which offered the best security also involved paying sizeable and ongoing performance penalties. One logical progression is to resolve these details at compile time wherever it is possible to do so. Since operating system dynamic re-configuration may well be far less common than periods of stable systems operation this seemed a promising direction.

Other system designers had similar ideas. One approach, as used in the Choices [Campbell and Tan, 1995] system, was to express the operating system as a set of object classes. The core of the operating system could then be constructed using specialised versions of these classes that inherited much of their functionality from the supplied parents. A less restrictive approach was used in the Flux OSKit [Ford et al., 1997] which used the general language concept of a library so that system components did not mandate a particular language or programming style. Both of these systems effectively used the processes of compilation as the mechanism that constructed a particular operating system solution.

The systems also relied on software mechanisms to provide flexibility to the interface between

custom code and provided component. The object-oriented approaches relied upon the mechanisms that are at the foundation of that design philosophy. Functional polymorphism, multiple interfaces and specialisation through inheritance allowed one module to have many interfaces. The Flux system used a component object model (specifically Microsoft's COM and some custom code) to provide an abstraction layer with similar, but language neutral, functionality.

The reason they needed this is, in simple terms, because the task is much bigger than it looks. It would seem that writing a module, say a memory manager, would be relatively straightforward. However we are effectively back at the initial problem, the construction of a universal mechanism, which in practice does not exist.

The module must be coded to allow for all possible ways in which a memory manager can be expected to operate. It must also be able to interact, efficiently, with an unbounded number of other modules many of which have not even been imagined, let alone designed or constructed. Including this flexibility in the relatively static construct that a library or object provides is far from easy. Even worse the complexity of constructing a module grows as a function of the number of modules that exist. In simple terms the more flexible the system attempts to be, and the more widely applicable, the more it suffers from an exponential growth in complexity, the same complexity that made the construction of a universal solution intractable.

4.7 The Idea of Auto-Generation

The previous approaches, while each limited in their own way, seemed like steps towards a worthwhile goal. The question was how could the strengths, primarily the structured approach to system design, be made more flexible without a cost in specific performance. Similarly how could the complexity inherent in both flexibility and performance, which was multiplied by the goal of crafting universal and reusable components, be restrained. The complexity of any solution, which would impair both use and the ability to generate reusable outcomes, was especially critical.

The first step towards crafting a solution for this thesis was to consider a base case. That is to say what is a minimum problem which an operating system could be called upon to address[5]. For example consider an operating system that does nothing but take readings from a sensor

[5]The C standardisation committee used a similar process with an elevator being the base case. This is why C I/O is in a library rather than integral to the language. The base case being used made it clear it would not always be needed.

and send them down an attached cable. Many of the proposed solutions, and existing general products, will include within themselves a massive amount of complexity that is not related to the problem actually being solved. As such they are both overly complex and sub-optimal as a solution. For many problems the base case may be solved without any structure that we would recognise as an operating system. A program running directly on the hardware could meet this challenge, that is the operating system functionality would be entirely subsumed within the compiled application.

The second step was to consider the expansion case, the pressures that cause an operating system to expand its capabilities. Our base case, even though it does not contain a separate software entity we would consider to be an operating system, is a complete computing system. The need to extend the system could come in several ways. New underlying hardware (or changes in a lower level software layer) would require changes to take advantage of new capabilities. Iterative advancements in the system or repairing bugs could cause the need for changes. The programs in the application layer which are the source of the system's value could also demand new mechanisms to support their operation. The important element was that the drivers of change would generally come from outside the area of responsibility of the system and be due to external pressures. Iterative change, such as bug fixing or optimisation, which could be considered entirely internal would still generally be driven by an external need for the change to occur.

Taken together these points enable us to see that the operating system can be given a strict, but not absolute, definition. The base case states that with no pressures there need be no operating system. The expansion case states that the system will expand in response to external demands from its environment which are outside its control. This allows us to define an optimal system to be one that represents a minimum connection, in terms of complexity and overhead cost, between the demands of the application and the resources provided by the hardware. Even highly system dependent properties, such as multitasking, represent the results of a demand from the application layer, specifically the demand to be able to run multiple programs. This has been recognised to some extent in the end to end argument [Saltzer et al., 1984] although it is stated as a force within modules, rather than as a single continuous connection between two invariant (from the system's point of view) extremes. The important insight is that this allows us to limit the complexity we must consider using in constructing our operating system to only

that demanded by the specific application under consideration.

There is also a very specific limitation on this approach. It assumes that the system will be customised to a relatively narrow, and known, range of application demands so that it can be tailored to their needs. This is a property which will be referred to as a custom system, as opposed to a general system (like Unix) which can make no assumptions about the specifics of the applications it supports. A custom system, of which the Magnus project is an example, is willing to specialise in order to gain a localised competitive advantage. It is also quite possible that this specialisation will have negative interactions, or at best represent unnecessary complexity, for some other set of applications. If the specialisation was truly superior in all cases, for all applications, then an improvement that should be integrated into the existing general systems has been found.

It can be seen that this definition divides two very different design goals. A general system is constructed to support the full breadth of application software, and as such is limited in how much it can assume about the operation of any given application. In the same way, since it can not assume the software will be written expressly for the system, there are limits as to how many demands it can place upon application authors if it wants to make the porting process economically attractive. This is why general systems focus on the application programming interface as being a neutral contract between two independent and loosely aligned system entities. It is also why, given the immense amounts of investment in application software for the existing general purpose operating systems, there is great temptation for new general operating systems to conform to existing interfaces (e.g. the POSIX standard [IEEE, 1990] derived from Unix) even though this limits their ability to innovate outside of the established operating system "box".

The special purpose operating system follows an entirely different logic. The entire system is constructed as a unified whole for a single common purpose. There are many reasons why this could occur, some need or constraint that dominates the design to such an extent that a general purpose operating system is an unacceptably poor match. At one extreme the system could be far too limited, the realm of embedded systems, while at the other end the required performance could be too demanding. The central advantage of such systems is that all components of the system will be built together and thus can be highly integrated to meet the design needs. Unlike general systems one component can safely make assumptions, and place demands, upon another part.

There are several restraints that work against the growth of custom operating systems as a field, and contribute to their under-representation in operating systems texts. One is that each system fills a narrow niche, is numerically dwarfed by the population of general systems, and is likely to be highly proprietary and expensive in order to recoup the costs of development from a small user base[6]. There is also the expectation, even if all system information was available, that any components generated as a part of the systems construction, to the extent they are customised, are made less useful for the construction of future systems which will have a different design imperative. In short custom systems can be seen as an evolutionary dead-end in terms of operating system theory, neither able to profit from or contribute to the development of operating systems as a field. The cost and complexity of construction for an entire system also means that niches capable of supporting development are rare.

At the same time there is value contained in a customised system that would be advantageous to recover. While the structure of the system may not make its components globally applicable they provide insight into how the design goal and environment shaped the architecture, design parameters and optimisations contained in the finished system. Being part of a finished system their performance within that context can be measured and iterative improvements contemplated. These lessons learnt will provide a good foundation for future systems facing a similar design environment. This may lead to families of systems or mechanisms based around a shared, but not global to all systems, design constraint.

There is an interesting paradox here. If customised operating systems could be more easily constructed, from reusable components, there would be more scope for experimentation and growth in this interesting domain. However customised operating systems are inherently badly suited to generating reusable components. Likewise reusable components, which must attempt to be general across all possible uses, are unlikely to achieve the focus and efficiency of one of the highly optimised components in a custom system. It is this challenge the thesis aims to address.

The solution this thesis puts forward is that the limitation of the operating system development ideas presented is due to the static nature of the solutions being proffered. The perfect system model, the minimum core model and the library / object oriented solution are all alike

[6]It could be argued that very low end embedded systems work somewhat differently and have substantial populations but low visibility and profit. Since their operation tends to be low value and require low performance they focus on cost efficiency and are not likely targets for customisation at the system software level.

in that their components are not expected to change in the process of application. The interface may change, the selection of particular components will vary, but the structure and operation of each component does not change. This is, after all, perceived as the basis of their value, that the one component is applicable across multiple environments. The weakness is that the components pick up some of the complexity from each of the possible applications, but cannot gain the advantages of specialisation because that would limit its general application.

An alternative solution is to have a way of expressing a general solution from which a context specific optimum can be automatically generated. The natural parallel is a compiler which can take an algorithmic description expressed in an easy to understand form and generate an efficient executable binary. The initial form contains a lot of detail the compiler discards and the executable includes a lot of optimisation and integration work the user does not need to worry about as long as it provides reliable behaviour. The project name, Shards, was intended to indicate this nature and that the pieces are not valuable in themselves (they are not fully formed components) but they are pieces from which a unified and focused object can be constructed. The important difference is that there is an automated, and controllable[7], method of transformation from general to specific cases.

The Shards approach is to have a collection of resources, which are effectively fragments that have been recognised as common amongst some subset of systems. These will be expressed as concepts that can be manipulated by the designer. Internally they will contain the code fragments (in an existing programming language) that would be used to implement this concept and some instructions on ways in which the fragments can be combined. These will be the input to a larger process which combines the system goal, the designers model and the implementation fragments and can produce a unified and customised set of components from these inputs. These components are then used as elements of the normal software build process for the target system. It is expected that both the "Shards" and the "glue" will be incomplete, and both require and allow custom code to make up the differences. This is how the system avoids the trap of requiring a universal system before it can be practically applied. It is also hoped that the process of generating custom code will reveal more opportunities to generate reusable resources and assembly processes. Each such creation is an expression of system mechanisms, in the case of resources, and system construction and tailoring, in the case of assembly processes. The

[7]In a way similar to the role of the preprocessor and linker in the language analogy.

argument is that the degree of control needed can not be contained in just the interface of a static module, even with the flexibility object oriented design allows.

The amount of control required will need a relatively sophisticated mechanism to allow this process of specialisation to be captured and automated. However this level of automation must be present or the approach will fall victim to the problem that directly writing the specific case is simpler for the system designer. The method cannot be a purely theoretical creation, it must be a tool the programmer on an actual development team will use while they work. In a similar fashion the use of an automated system, as a process that spans many projects, makes it far more likely that the outcome of a given project will be reusable. The automated system will assist with reuse, integration and specialisation of a general component to project specific needs without requiring deep understanding and modification of the module's internal details.

> "The complexity of software is an essential property, not an accidental one. Hence descriptions of a software entity that abstract away its complexity often abstract away its essence." [Brooks, Jr., 1987]

The complexity of operating systems is not incidental; it is a requirement in order to meet the tasks for which such systems are constructed. Nor is there a "silver bullet" which will provide orders of magnitude reductions in construction complexity. The process of seeing commonality, and expressing it in a reusable form, will still require a human intellect, inspiration and a substantial amount of work. But providing a common mechanism in which the solution can be expressed, a system of systems as it were, would still provide a unifying foundations for these individual advances. The ultimate goal is not a solution, per se, but rather a way in which the current problems can be clearly stated and solutions shared and compared. The aim is to create a positive feedback cycle where any advance becomes the base on which further advances can be made.

The selected approach, based on capturing and structuring the generation of an operating system solution, has several desirable properties. It has the possibility of being complex in its internal operation but practical in its application. Taken as whole the system could contain a great deal of flexibility but component selection and tuning could reduce the complexity when considered for reuse in a specific context. However, the practical issues of how the Shards software would be structured and built, which will be the topic of the next chapter, contained

challenges of its own. In addition, like most system software, evaluation had to wait on the system being largely functional.

Chapter 5

Shards Overview

This chapter will focus on the mechanisms that will allow the Shards system to be implemented. The goal is to develop a modular representation of system mechanisms. The implementation method used to represent modules are referred to as filters. These can be considered the framework needed to encapsulate functionality. In this chapter filter families, filters and transforms will be discussed. These all use the same method and structure but each represents a different conceptual level.

The Shards system is both the theoretical and automated process of applying filters and supporting their processing. The input to the system is information representing the customised system being constructed. This information includes the selection of filters to be used. The filters react to the information provided about the system and generate tailored output. The output is in the form of components, such as code or object files, that can be used in the construction of the system.

The input and the filters will vary depending on the needs of the project. The output is a reproducible product of the selected filters and provided input. Since the selection of filters results from the contents of the input, that input can be considered a concise expression of the system in as much as Shards is involved (it may not be generating a complete system). The single mandatory and central part of the input is called the project file and is used to trigger the process.

The process can be considered along the lines of compilation (with the project file taking the part of a main() routine in C). The information in the input is at a reasonably high level and does not specify the detail of mechanisms much as a line of C code does not specify into what

it will be compiled. The filters selected manage the conversion into a form that can be used in the system build process. Each filter family represents a mechanism or piece of functionality but it is also the process by which it is tailored to the design goals of the system as contained in the input and as determined in the process of system construction. A complete Shards process may require many filter families, all of which will operate in turn and combine their efforts to generate output.

This approach allows the Shards system to provide a concise description of the target system in the project file. The Shards process is a complex but automated sequence in which the selected set of filters process and extend the information that was introduced by the project file. The output is a solution that represents the input, the filter processing, and the mechanisms the filters represent and contain. This output will be in the form of source code which can be directly used in the build process for the target system. This approach provides a very flexible framework that can be used to describe and construct a wide range of systems using the same process (and possibly many of the same filters).

The discussion of automation mechanics will take up much of this chapter but it should be remembered that this is just one possible solution. The true value comes in thinking of system construction as an automated process and supporting the use of modularised and goal-optimised functionality as the foundation of building new customised systems software.

5.1 System Construction

The construction of system and operating system software is currently considered the same as any other large software. Source files are written in a bespoke manner as the software must be optimised to the design goal of the system and is not expected to be directly reused. For the same reason, being able to reuse software libraries from previous projects is much less common than is the case with application software. This situation could be represented as in figure 5.1. In this diagram, a range of source code assets are integrated by the compiler into an executable system.

The source code is organised and compiled using common build tools. As a large software project with a potentially high variation in targets, there is likely to be substantial complexity in the build process. There is also likely to be duplicated code optimised for different hardware platforms and configuration options which may be integrated in the process of compilation.

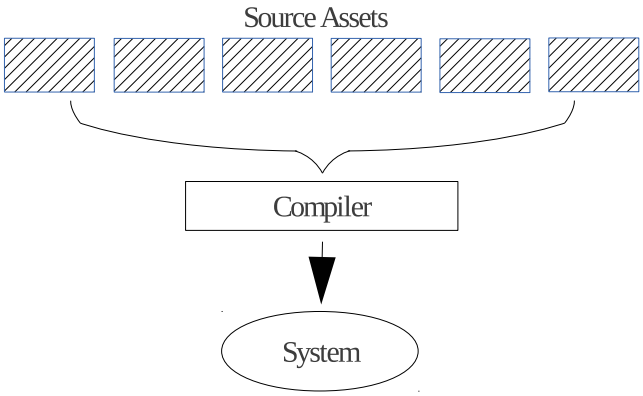

Figure 5.1: Traditional System Generation

Large operating systems like Linux and Windows rely on modular code that can be dynamically loaded as required while the system is running. This reduces the need for users to recompile the system to suit their environment or for distributors to ship many varieties of the core system. This approach does not solve the complexity of supporting many different target systems. The number of loadable modules can grow to be quite large and the resulting permutations that require testing and support are an exponential function of the number of loadable modules.

The result of the compilation process is an executable file. This is once again similar to the process followed in the building of any large application. For an operating system the difference is that it will have a boot loader of some sort so that it can start when the system boots, place itself into a privileged mode where it has access to all system resources and take control of the capabilities provided by the underlying computer resources. This will generally be a hardware platform but virtualisation, firmware and architectures with many software layers can mean that the system is in practice running on another layer of software, a virtual machine. In practice the difference is not important, it simply provides the foundation on which the system rests.

One result of this approach is that making fundamental changes in the operating system requires modification at the source code level. There are unlikely to be specific tools to make the process easier other than general code visualisation tools. The complexity of the system, the high degree of coupling in the source code, the fact it is specific to the system being studied and the complex run time behaviours and interactions, mean a great deal of study and understanding

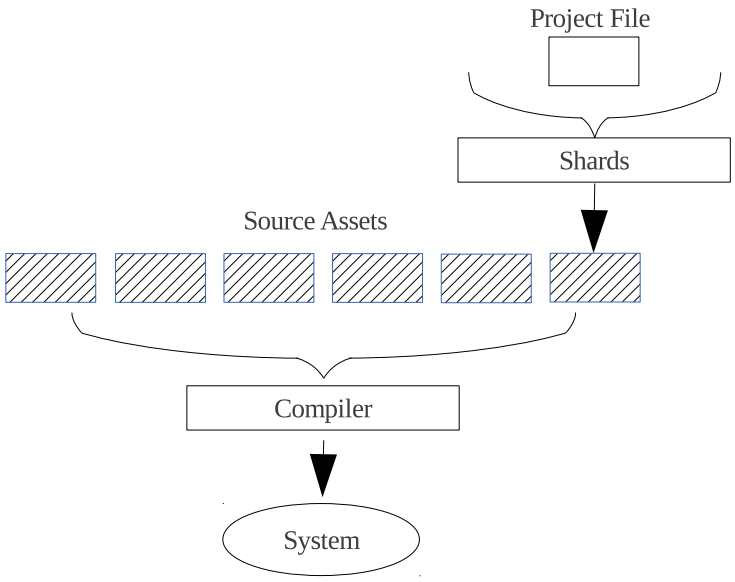

Figure 5.2: Shards Extended System Generation

is required before productive software modification can occur.

5.2 Shards Method

The core extension added by the Shards system is enabled by a single change in the order of system construction. This is an additional level of processing before the normal build process. This provides a level of indirection between what is constructed by the system designer and what is presented to the compiler. Some or all of the inputs to the process of system building may be instructions which the Shards system will resolve. The Shards process will run and generate code or object files which can be presented to the compiler as in the existing methods of system construction. This is depicted in figure 5.2 which shows a single asset being generated by the Shards system prior to the system construction step as in figure 5.1. The Shards system could also be responsible for generating all of the source needed for system construction. This would depend on how much functionality is being customised or reused.

As indicated in the diagram, blocks that can be presented directly to the compiler, and which do not require manipulation by the Shards system, can continue to exist and will operate as

normal. The advantage is the Shards system can slot into an existing system without taking full responsibility for complete system construction. This allows the Shards system to co-exist with existing system construction methods, to be added into existing projects and to only be used where it can add value. The elements for which Shards will be involved will be noted by the designer as entries in the *project file* which is a structured text file the Shards system will read and use to guide its actions.

The project file contains two types of information. One type of information defines *environment variables* which represent a statement by the designer. For example, this could be a data item that identifies which sections of the system must be generated by the Shards systems or some context information about the system in which it will operate. The other type of information are *filter families* which are the mechanism involved in generating the needed sections. Both of these data items are optional. A Shards system with an empty project file will do nothing, generate no output, and the system can be compiled as normal if it is not dependent on there being a component created by the Shards process. As the Shards system does more the output it generates can be integrated into the build process to replace or extend existing code. The environment variables can be considered input information or triggers for the filter families specified. This is intended to allow a filter family to be reused in multiple projects. If the filter family does not need, or does not query, data in environmental variables then they are ignored and their presence has no effect.

The internals of a filter family and the context in which it runs will be considered in more depth in section 7.3. At a high level it is a name that takes a configuration argument. It is called a filter family because it represents multiple internal filters some of which may be shared with other filter families. These have been selected and ordered such that they will absorb the configuration argument, optionally query environment variables as required, and generate output for integration into the system through the process of compilation.

The output links to existing code through the normal mechanisms of the language such as function calls. There may be a degree of variation here such that the application programming interface (API) provided by the output does not naturally match the names or manner in which they are called in the existing system code. This is addressed by using small and simple filters, which may be specific to the project and which map the calls as needed. Alternatively if the Shards component that will be integrated is known at system design time, it is expected the

function calls it provides would be used directly.

The Shards system can also trigger the compilation process as part of its execution but there is no real need to link the two. The Shards process will be fully complete before compilation and the code presented to the compiler is like any other code from the point of view of the compiler. As auto-generated code, it should not be human edited because it is probably less conveniently structured but also because any changes will be lost the next time the Shards process is run. However, this is not in any way enforced. The designer could execute the Shards system some number of times during the development process and perform refinements and fixes as normal code after that point. This also explains why the Shards system does not consider low level code optimisation. The Shards process defers to the compiler to perform that step.

The output has been described as auto-generated but in practice it is expected to be more about selection from source code resources than complete auto-generation from some primitive form. For example consider a filter family which deals with a particular memory management strategy applicable to Unix-like systems. There can be expected to be several variations which may be stronger or weaker depending on the needs of the system and the intent of the designer (as expressed through selection of the filter, configuration arguments and environment variables). At the same time, since these variations are closely related, much of the code may be common. The filter family will generate, configure or modify the code that is specific but will simply select from code that is common. Since common code can be included directly as a resource there is no point in generating it. The code can simply be written directly by the individual who constructed the filter family which the filter incorporates directly into the output. As such much of the complexity in the filter is expected to be in organising available resources rather than performing complex auto-generation of large blocks of code.

The existence of variations within one approach is itself a worthwhile area for research. If it is found that one implementation is sufficient to represent the needed functionality in all existing systems then this suggests that there is little variation across systems. The filter may need to do nothing more than output the code representing this common implementation knowing there are no practical alternatives. If another alternative, perhaps for a new system environment, is discovered then expressing the points of difference is the purpose of the filter. A later researcher might determine that two variations can be expressed in a unified form capturing the best parts of both and thus reduce the number of variations. The filter becomes a way of expressing all

known and compatible implementations of the functionality it represents. This can also include configuring values used to tailor one implementation to optimal performance in the system in which it will be used. Stated another way, a variation may be as simple as changing an internal variable to modify performance or as complex as containing multiple implementations using different algorithms and approaches to provide the same functionality.

The system designer can also use arguments to the filter family in the project file to explicitly control which variant the filter family will generate. The filter could also make internal tuning parameters available to be explicitly set where this makes sense. This allows the designer to override the automated selection and easily switch between variations during system construction to examine which performs better in practice. Testing the different variants can provide feedback on which variations are valuable. This information is useful to the system designer and can be used to improve the method used to select and configure implementations within the filter. Encapsulating functionality within a filter provides a structure to research in the domain. It provides a single mechanism that can gather multiple approaches which are mostly similar but optimal in different environments or under different design goals and assumptions.

5.3 Componentisation

The advantage of the Shards system is that these variations on system mechanisms are all contained within the wrapper provided by the concept of the filter family. These filter families can be used as part of a construction effort even without full knowledge of all the detail and complexity they contain. The filter family wrapper allows a general system concept, such as memory management, to be named and automatically bundled with a number of implementation variants. These variants are sufficiently similar that with the Shards process doing some selection, tailoring and code generation, they can be considered inter-changeable parts of the higher level design. They can be described by their shared purpose and documented in terms of their common interface.

This allows the mechanism to have some degree of boundary between it and the system in which it was created. This boundary does not exist as a run time construct, and exerts no run time performance penalty, but exists through the mechanism being expressed as a Shards filter family. This takes some effort, as opposed to simply writing the code directly, but there are a number of potential advantages outside of the immediate system being constructed.

The first benefit is that a filter family binds a name which identifies the filter family. This name is constructed in terms of what it does within the system and how it differs from incompatible variations providing similar functionality (since sufficiently similar variations could be integrated into the filter family). This name can be used as the identifier of this role, the included implementations and the steps need to apply it as part of the Shards process. This allows for the productive development of a taxonomy of system concepts. This taxonomy will also be informed by the implementation. If the implementation of the filter family indicates that two seemingly similar concepts have extremely little commonality, then they should be expressed as independent filter families. This informs the taxonomy that similar concepts may have more difference in practice than appears at an abstract level. The reverse is possible as well with two filter families being discovered to have such similar mechanisms that they can be merged.

Another advantage is the potential for reuse. If the concept has a functional definition and a boundary which also provides a programming interface, then it can be more easily be removed from the system and reused. This will be a central concept in providing value to the use of the Shards system and will be discussed later.

The possibility of reuse allows for taxonomies and filter families to survive the systems that created them. For example, consider a system like Multics that is far from novel or cutting edge and probably not optimal in today's computing environment. It is primarily remembered as the predecessor to Unix and is no longer actively used. Attempting to understand one of its subcomponents such as memory management, may still be interesting for the purposes of operating system history, tracing operating system evolution or to provide some insight into the domain that could be reused in the design of a more modern system. With the mechanisms of the system contained only in the end product of raw source code (on obsolete and unavailable hardware) and the memories of the original designers, recovering that information will take so much work that even if it were possible, there is likely to be little interest in doing it. If the memory management strategies were captured as a structured logical component which could be isolated from both the larger operating system and such issues as the programming language and the underlying hardware, it would be easier to understand, compare with other systems and look for any unique value for possible reuse.

A focus on modular components allows the field of system development to become more than a succession of systems in which only the highest level of abstract concepts contribute to the

field. Instead the taxonomy, the functionality and the historical implementations might all be able to add to the depth and capacity of the field of systems design. The iterative evolution, the failed experiments and even the imaginative solutions that came before the technology or market was ready, could be captured and stored for future use. This vision of the field of system development is extremely ambitious and is well beyond the scope of this document, but it is the motivation for developing the Shards system, which provides a suggestion of the framework, that may make it possible.

5.4 Shards Mechanics

The Shards process is an application closest in function to a compiler but operating at a higher conceptual level. It does not need to be optimised for efficiency and security because, like a compiler, it operates as part of the construction of the operating system and has no run-time presence. This section will give a high level view of the mechanisms which make up the Shards system. The detail of how this is implemented will be discussed in Chapter 7.

As mentioned previously the project file is the initial material for the Shards process and represents directions from the designer on how it should operate. The information in this file can be quite concise, abstract and high level because the components activated by being named within it encapsulate much of the complexity as part of their operation. The example of constructing a Unix-like memory management component could easily be performed simply by naming the filter family known to generate that component. This would be enough information for a project file if that was the only requirement.

The ability of the filter family to provide additional value depends on it being able to determine the needs of the system it is constructing. The easiest and most accurate way to gather this context information is to rely on the system designer to provide it in a form that the filter family recognises. For example, if the filter family provides several different implementation possibilities, the designer could provide an argument to the filter in the project file. This could restrict or select the implementations the filter family will consider using for generating the requested component.

The more complex cases come when the filter has a number of possibilities to select from and attempts to determine the optimal selection automatically. The filter can be programmed by its creator to gather information from the environment and calculate which implementation

is most appropriate without the explicit direction of the designer. The simplest examples are the environmental variables contained within the project file. These are variables describing some element of the target system. For example, the designer may know the target system will have a standard amount of memory and state this in the project file. This information is not directed to a specific filter or with a specific outcome in mind. A filter may check whether a value relevant to its operation has been specified and make use of it. Continuing the example, a filter family implementing a memory manager may find the size of memory specified favours some implementations over others and use that to determine which is used in generating its output. It could also use this information to tailor the implementation through rewriting sections of the generated code to include the environment variable.

A result of this communication between the filter family and environment variables is the value of a shared vocabulary for system description. If all designers and all filter authors have a common notation for describing system attributes then communication is more productive. This would allow a system designer to know which values can be recognised and provide definitions, if important and known, without considering the specific nature of the filters being used. This allows filter families to be swapped in and out without having to redefine the attributes of the system. It also allows the filter authors to know what data may be available and the syntax with which it will be expressed. Developing this syntax would itself be an investigation into system domains and what values are useful for description within that domain. However the generation of such material would be a creative human process, likely to be ad hoc and iterative, and as such is well outside the focus of this thesis.

A filter family may also add, remove or modify environment data as part of its operation. This allows different filters to communicate. For example, consider a memory management component that wants to claim some memory for metadata, caching or some other form of optimisation. To reflect this change it might modify the value describing the amount of memory available on the target system to represent this change. Alternatively it could create another environment variable to indicate the reduced amount of memory actually available to the rest of the system. The advantage of automated filter interaction is that these operations do not require the manual intervention of the system designer. It also allows the possibility of cooperation between filter families. A filter family supporting processes that are very sensitive to memory latency could be written to make use of information made available by the filter family that is creating the

memory manager.

A filter can also use information not directly provided by the designer for that purpose. For example, the project file will specify one or more filter families which indicate the designers intent for the Shards process. This information is also available to the filter family when it executes, allowing it to see which filters have been run or are scheduled to be run. Continuing the memory manager example if a filter family supporting memory latency sensitive processes recognises the filter family providing the memory manager, it can derive information from this fact. As before, there is the possibility of intentional creation of cooperation and synergy between filters families which have overlapping functionality or areas of interest. It is also possible for the cooperating filter families to add additional occurrences of themselves in the execution order. This would allow them to interact over a number of processing steps.

The Shards process itself uses this information to trigger each filter family in turn. A filter family collects and encapsulates all the filters needed to represent a particular system concept or functional segment. It is expected that the filter family will add a substantial sequence of filters to the execution order to carry out the processing required. It is ideal if the filter family focuses on scheduling the filters needed rather than doing processing itself (other than perhaps some initialisation). This allows the structure and operation of the filter family to be defined in one place and easily observed for what steps and components are used to automate it. The filters used should also be as concise and focused in their function as possible, ideally representing a single operation, calculation or modification of state. This allows individual filters to be easily understood, modified and reused where common operations are observed.

The filters can be considered as being similar to expressions within a high level programming language. They can also act in a way similar to the control structures of a programming language. They can add additional filters to be executed after them, they can skip over filters that would have been executed, repeat from a previous point in the order of execution, and they can use environment variables to communicate. Environment variables include the name of the filter family for which they are intended, easing this form of communication and reducing the risk of name collision. Communication occurring entirely within a filter will use programming language variables as normal.

It is expected that most filter families will not use extremely complex controlled execution orders. Most filters can be expected to have some initial filters that gather information and

express this as filter family specific environment variables. These will be used to make a decision which is generally expressed as a sequence of filters. These filters will do the work of generating and tailoring the output. The output could also be input to a later filter family but will ultimately involve the generation of source code which is the requested component needed for system construction.

There is also a category of filters known as *shim* filters. These are project specific filters which exist to do translation between filters with incompatible interfaces. This would occur when two filter families express the same concept, such as how much memory the system has, in different ways. The ideal solution is to have a universal syntax such that all filters use the same notation. The practical solution is to write a small filter that translates the information such that both filter families can work together. This means that the Shards system can be used and progress can be made during the iteration towards shared standards of expression. A shim filter should restrict itself purely to this translation role and not include aspects of component generation so that they are easy to write and safe to discard when interfaces change.

All of the information within the Shards system is stored in a single data structure. This could be a list or a tree but is currently implemented as a list. This is because the focus is the logical order and content of the data rather than the syntactic structure which is the focus of a compiler. The syntactical correctness of any code will be ultimately be checked by the compiler to which it will be presented for construction. This means there is little value in the Shards process reproducing this process. Each element of the list is a tuple representing a single data item. The first element is the *head* of the tuple which always consists of three variables. The first element identifies the filter family with which the data is associated. This is not exclusive as filters can create and read data associated with other filters if they wish to cooperate. The second element of the head identifies a data type, defined by the associated filter family, which can be used to understand the intent and structure of the entire tuple. The final data item in the head may be a subtype, identifier for this specific item or first data item as defined by the structure of the type. The remainder of the tuple can be arbitrarily long and will follow the structure it has identified in the head.

The list of tuples is referred to as the *data chain*. The first section contains the list of filters and their arguments and is referred to as the control chain. This is initially assembled by parsing the project file created by the operating system designer. The Shards system will iterate

through the control chain and call each filter with any arguments provided by the project file. The executing filter will be given the data chain as input. The arguments are used to make it easier to find commonality in the operation of filters so that reuse is easier and fewer discrete filters are required. This also allows the designer to provide arguments which modify the filters internal operation for testing or special applications.

The control chain is itself part of the input and so a filter is capable of adding to or modifying the order of filters through changing the data as it passes through. In practice many changes will be performed through using function calls provided by the Shards system. This allows the operation to be more reliable, reduces the need to manually modify the data chain and also allows the system to track when such operations of interest occur. The API also provides utility functions such as searching through the data chain for tuples of interest which is a common filter operation.

It is expected that some filters will extend the data chain with information that can be used in determining the optimal construction of the requested system components. This could be information based on performance analysis or source code that represents some part of the target system. It could be either generated by the filter or parsed from provided files.

5.5 Filters

The heart of the Shards system are filters which are organised in series to do the work required. A filter can be considered as both a mechanism and a logical construct. In terms of mechanism a filter is a wrapper and support for user supplied code in a programming language supported by the Shards system (currently C++, but previously Erlang). The advantages of adapting a well known and mature programming language are substantial. It provides a proven language, compilation and support tools and most importantly, existing familiarity for potential system developers. While a custom notation was considered for expressing the operation of a Shards filter, no significant advantages were found. The internal operation of a filter and manipulation of data, did not require novel programming language constructs to be expressed. The Shards system provides a template into which the code can be placed so that it can be integrated into the system and called as a filter. The Shards system also provides an API through which the code can gather data from the environment and make changes to the Shards system such as adding filters, modifying environment variables and various other capabilities. Some headers for

this API can be found in Appendix A.

The suggested model for the internal structure of a filter is that of a finite state automaton (FSA) which also mirrors the traditional model of operation within a compiler. It will be presented with the data chain and encouraged to use that as an input stream, a trigger for actions and storage for intermediate steps in processing so that filters are loosely coupled. There are also API commands to write to environmental or shared variables when the information is not connected with any specific point or components in the data chain. The filter should respond to the data available and transition through internal states which represent both a response to that data and steps towards an output (which in turn becomes input for later steps). Use of the API to change state, to modify filter execution order and read or write environmental data are all tracked and mapped to a point in the processing of the data chain where they occur. This allows the execution of the Shards system to provide integrated support for monitoring, analysis and debugging. This encourages filters to make the intent of their processing clear and to keep the scope of an individual filter small which enhances the ability to reuse filters.

Once a filter reaches the end of the data chain, it should respond by completing its processing using an API call so the next filter can be called. Some filters will rewind the data chain so they can apply changes based on analysis gained from the initial processing of the data. When there are no filters on the control chain remaining to be processed, the Shards process is finished. Whether it has finished successfully will depend on whether errors or omissions are detected during the ensuing attempt to construct the system using Shards generated components.

Filters should by nature be passive. A great deal of the data in the data chain will be of no relevance to its goal. This is required because the Shards system is unable to make assumptions about what functionality each filter represents. As a result, it has no option but to give it access to all the state and progress data it has. Since this data represents all communication between all filters much of it will not be intended for or meaningful to any specific filter. The correct action for a filter is to ignore all data passing through unless it finds a match it recognises. A match represents a pattern in the data that the filter recognises as being the starting point for the transformation it embodies. This transformation can result in a difference between the data read in by the filter and the data it outputs as well as potential changes to the global metadata and the filter order. When the filter has completed its operation, or processed all the data in which matches could occur, it can then indicate to the Shards system it has completed its work

and the next filter can be executed.

There are various types of filters. Filter families and some filters are primarily concerned with selection of filter order. When called, these filters may examine the environment, make some internal calculations, and express their result in the decision of which additional filters to add to the execution order. It is also expected there will be some filters that contain relatively concentrated processing which is mostly easily expressed in the underlying programming language. These will read in collected data and generate a result to place on the data chain. This result will generally be a trigger for later filters to operate upon.

The term *transforms* is applied to filters that provide a single conceptual step in processing. It can be considered a basic conversion, change in data or state, or step of processing that cannot usefully be decomposed into smaller functional elements. These may represent frequently used operations that can be treated as procedures and made available for use in the construction of sequences by future higher level filters. Common models of filter construction and a rich pool of transforms can provide a foundation to assist in the construction of new filter sequences.

Filters are ideally small, containing the minimal number of internal processing steps and states, logically atomic, and accept arguments which are intended to encourage the detection of opportunities for reuse. In other words the operation it performs will represent a reasonably concise change such as finding a particular pattern in the data and performing an action. The idea is that if the filter could be devolved into a series of cooperating sub-filters then it generally should be. Smaller filters have smaller scope and simpler internal operation. It makes the filter easier to understand, modify or reuse and means it is easier to express meaning through reordering the sequence of filters. Long, complex or variable sequences of filters can be gathered in a filter to manage the addition of the sequence to the process list. This approach of lower level sequences being gathered into management filters which represent higher level concepts continues up to the family filters. Family filters are unique only in that they occupy the top of the concept hierarchy.

It is also possible that some filters can themselves be generated using the Shards system. This concept is referred to as *meta-filters* and would involve a step prior to the execution of the wider Shards system. This step would take a concise notation for common operations and generate filters to perform the required operations. The advantage of using meta-filters is that it allows the iterative development of a specialised language for describing filter operations. There

is some value to this as the internal operation of many filters is very simple and repetitive. Explicitly programming this within the structure of a Shards template can be somewhat tedious. In practice, however, it does not offer additional functionality and can be replaced by reuse of appropriate designer created filters so it remains a future possibility rather than a core requirement of the Shards system.

In combination, these filters allow the generation of a system component to be requested. The top level filter family will create lower level filters to determine the appropriate process required. As the focus moves down towards the low level operations, the scope of the filters will become more limited and focused on data manipulation, increasing the possibility to reuse transform filters as elements of the process. It is expected that initial filters will tend to add information to the data chain and environment variables while later filters will convert this information to files suitable for use in construction of the system. Since the core of the filters is a full featured programming language, the system is extremely flexible as it allows a wide range of potential approaches and applications. Indeed, the Shards system has been used to generate web pages and could in theory be applied to any processing of structured data.

The operating system designer uses filter selection as the primary way to describe the structure of the component, or systems, being built using the Shards system. Ideally, the designer would be able to select from pre-existing filters and use execution order, environment variables and filter arguments in the project file to adapt them for use in the project being designed. The creation of some small shim filters might be required to connect filters using different notations from the project but these will tend to be simple. The ability to reuse a filter, which contains within itself the code for a system component, offers the potential for a reduction of effort in constructing future systems.

If the component generated automatically by the filter can be reused in concert with the design goals of the new system, then this represents a saving in development effort. This allows the focus of effort to be on those components which are unique and novel to the system and thus will require new code, new filter families or an extension of an existing filter family. For a customised operating system, it is these novel elements of the system that are vital in generating commercial value and making the project viable.

These new or modified filter families naturally extend the depth of the resources available to future implementers creating a virtuous cycle. A pure research project might focus on isolating

and encapsulating existing system functionality into the format of a filter. An experimental research project might focus on the creation of a new component to test out a new theory about system construction or operation. In both cases the filter family could be developed in isolation or slotted into an existing system for testing, knowing it could be more easily integrated into later system construction efforts. This would help make operating systems more practical and productive as an avenue of pure research where there is generally not the commercial motivation or support to construct a full system.

Expressing a new component as a filter family allows it to be used as a component in a system "recipe". At its simplest, the system recipe can be considered as the project file or the initial control chain that results from it. This recipe can be cleanly expressed, understood, modified and replicated by others within the context of the Shards process. This enables the structure of the Shards system involvement in system construction to be easily transmitted, tested under diverse conditions and compared. This allows for much easier experimentation and incremental advance in the domain of system research and implementation. This process and way of thinking about system construction is the true contribution of the Shards system. Ideally, a theory or mechanism will be matched with a reference filter family that both demonstrates it and serves as a base for more specialised variations of the same approach. This allows theory and implementation to be synchronised, just as a theory of programming languages would be considered unproven without a reference compiler. This would also allow implementation details and test results to inform and drive the development of theory.

5.6 Application Load Analysis

The Shards system has been introduced as a process and mechanism for capturing the process of system construction. The use of filters allows system mechanisms to be packaged in a form that can be manipulated and assembled in an automated fashion. This approach can be used as a foundation to which extensions can be added. One path of extension is increasing the amount of information filter families can use to tailor themselves to the specific needs of the custom system they are producing. This has already been introduced in terms of environment variables which represent the designer or a filter announcing some attribute of the system to other filters.

There are a lot of possibilities in this direction. Filter construction can show the need for specific system information to enhance optimisation. The design of a customised system or an

invented scenario may suggest new ways the system environment can be expressed in a form filters can use.

A less obvious example is what will be referred to as the *application load*. For some custom systems, the applications that will run on the system are known in advance and crucial in giving value to the system. A Martian probe like the *Curiosity* rover or a high-speed share trading system will have a very specific range of software they run and will run this software for long periods of time. The system is given value by the combination of its elements and a very specific goal rather than being a general purpose computing environment like a desktop or smartphone. Yet even in a modern smartphone some elements, such as the radio module, are likely to run a single piece of firmware for long periods of time.

This aspect of customised systems can be compared with the dominant model of *general systems*. In the general system the services that application software will require cannot be known with precision. This is because the domain of application software is extremely broad and some of the demands are contradictory. It is also not static as new applications are constantly being created either for new functionality or to take advantage of the evolution of computing hardware. In such an environment, it is sensible to create a clean internal architecture (to ease future extension) with a logically complete interface that can be published as a target for application software to be written to. The application software, which knows its own goals best, can then adapt this general API to its specific needs. The Unix API is an example of this approach, it is so well understood and known that it has become the basis for a standardised interface that is supported by multiple general purpose operating systems [IEEE, 1990].

If the entire range of software that will run on the system is known in advance, is available in source form (which is likely since it has been written for the system) and gives the system its value, then it provides a definition of what is optimal for system operation. Whereas a general operating system will focus on the broadest and best average result from optimisation, in the presence of a specific application load, a custom system has the potential to do better for the specific design target being focused on.

There will always be some potential for sub-optimal interaction between a standardised API and a specific application set and system goal. In most cases, this will be ignored as the cost of modifying an operating system is very high. The estimated improvement in performance and how that increases the value of the application would have to be very significant for it to

be considered. The Shards system is explicitly based around the construction of customised operating systems. As such the more optimisation advantages that can be captured the more viable the system construction effort is likely to be. The Shards system should be considered a complement and not a replacement for general systems.

It is also possible to pursue a hybrid approach. In this case the general API is available for low value or rarely run applications on the system. The applications for which performance is critical and from which the system derives its value are gathered as the application load. While they are no longer the totality of the software the systems runs, they are the applications which should drive optimisation decisions and API extensions in the generated system. Stated another way, the point of the application load is to focus attention on the high value applications within the system.

The application load is ultimately a selection of source code. Within this code there will be a lot of internal structures and a number of interactions with entities outside of the program. These are generally referred to as system calls but could also be to libraries or other supporting software systems. These interactions can be implemented through a number of methods including the traditional function calls, messages, events and interrupts. All of them can be summarised as passing a data structure to an external system component in the expectation of some service being provided.

The Shards system does not provide an automated method to make analysis simple or automatic. Deriving useful information from a body of source code is complex. One way to make it more manageable is to accept that there are limits to how sophisticated the analysis, especially in early versions, can be. This is acceptable as the optimisation process does not necessarily require complete understanding of the source code to derive usable inputs.

One method is to use custom or existing code analysis tools. These tools may do a static analysis of the source code and attempt to understand its structure. Other tools might instrument the code, execute it, and collect information reported back from its operation. In both cases, the tool can be expected to emit a report containing the results of its analysis. This information could be parsed by a Shards filter, which understands it, and relevant information added to the data chain. This would allow filters to examine this information and consider whether it will inform their construction of the system component. In this way, Shards as a foundation can be used to integrate existing and future efforts in code analysis.

Another way is to use the Shards process itself. The source code can be parsed and relevant sections added to the data chain as tuples. It would be possible to use an existing compiler for this purpose and construct the data chain using the compiler's intermediate format. This is not done because the compiler will discard some amount of information (for example, Shards filters may react to keywords in comment blocks or use extended forms of the language) and intermediate forms are not standardised. Instead, the initial filters are more likely to reuse parts of existing compiler front ends (such as LCC [Fraser and Hanson, 1995]) to convert source text into an equivalent form that is more structured and easier to process.

The Shards system is not an ideal platform to do static analysis but it can derive information about the context and rough order of system calls within the code. Even with only a basic block structure, order of calls, call arguments and immediate context, it is possible for filters to derive usable information. Likewise, some simple statistical analysis, such as how often a system call is made or which calls do not occur, may provide information on which system operations to favour when there are optimisation choices.

Once established this process can be automated. The system designer will put a great deal of thought into investigating, integrating and acquiring tools that can generate useful inputs. This will be matched with filter families that know how to use this information where relevant to the design goal of the system. However, once constructed, the application load could be extended through updates to existing applications or additions to the number of core applications. The system could then be regenerated using this new application load with an expectation that filters will still behave intelligently and optimise for this new context. This is possible because the system is looking at the composition of the source code rather than the specifics of its operation. This means that the absence of the system designer or someone of equivalent skill does not block the system from evolution over time. As with any system, drastic change will call into question original design assumptions and provide unused possibilities for new optimisations.

An example optimisation case could focus on a single concern like memory management. Memory is a valuable resource, but not limitless, so operating systems must attempt to optimise its use. A simple interaction will involve a process requesting more memory (a get operation) and some time later releasing the memory back to the system (a put operation). The process cannot optimise this access because it doesn't know the global system state. The operating system can know the global system state but it does not know the usage patterns of the individual

applications or the system owners strategy for how to balance the competing demands upon the system memory. Gathering the global state also requires computational effort and can scale badly as the amount of resources under the control of the system grow. Systems will avoid doing so unless they know they need this information. As a result, operating systems depend on heuristics (which may allow for some control by the system owner) on how memory should be managed to try for a generally optimal solution. Inventing, analysing and improving the behaviour of these heuristics is a complex and actively debated topic [Wilson et al., 1995].

The Shards approach is based on the observation that much of the complexity is because the operating system has such limited information to work with. This is because the API it provides is so narrow in the interests of being concise and reasonably simple to learn even for people with no interest or knowledge of the underlying architecture. It must also attempt to optimise over the global scope of all possible applications. In the Shards system the application load is known and this allows the potential to do usage analysis as an input to generating the operating system. This analysis could also be combined with modifying the application so that the calls it makes provide the operating system with additional context information on which to operate. For example, an existing call to the operating system could be changed to a more specialised variety of the same call either by the developer or through automatic analysis of the application load. This would allow the context of the call, information gained from the application load and the design intent of the system being constructed to be integrated into the application level. The compiler would then process and integrate this more specialised version of the system call as any other code.

The end result will be an operating system and application load designed to work together. This is derived from the inputs to the Shards process which include the application load, designer input in the project file, the list of filter families and their order of execution. The advantage is that the analysis of the application load and integrating the results into optimisations in the operating system can be automated. An operating system designer could potentially add a filter family that will perform the analysis of the application load they provide (including filters to recognise the language in which it is written) and do nothing more. The analysis will gather what information it can and filters that can use the information will look to see if it is there and react accordingly. From the point of view of the system "recipe", the filter families given in the project file, this is largely invisible. It does not add complexity to the project file that limits the

ability to provide a concise description of the system. This is acceptable as long as the process is reliable in the behaviours it generates. The process also does not consume precious designer time on optimisation concerns that may change as the design of the operating system evolves.

5.7 Source Code Markup

The application load provides a source of information which filters can use to inform themselves as to the context in which the system is called. It is in effect trying to estimate the intent of the application author. This is difficult and lossy as the author's thoughts are not explicit in the source code and have been translated into the limited syntax of the language being used. The Shards system is not a compiler or tied to the structure of a given language and thus has more flexibility in this regard. The filter mechanism allows for the possibility of additional information being placed in the source code. This information more directly represents the authors intent and will be consumed or translated before a compiler can see it. It is in effect the author communicating their intent directly to a filter family that has the potential to productively act upon it. It is referred to as *hinting* in this thesis.

There are a variety of ways to provide hints. One of the simplest is to put keywords (with some format to make accidental hinting unlikely) into comments at appropriate places in the source code. This can trigger responses in filter families during processing. The advantage of this approach is that the source code remains legal even if presented directly to a compiler. It does, however, have limitations in how tightly it can integrate with the structure of the code.

Another alternative is to use operations that are not part of the syntax of the target language. This allows complex behaviours to be integrated directly into the syntax of the code. These could be operations unique to the needs of the target system or perhaps derived from a specialised language, for example parallel algorithms in Occam [Hull, 1987] (or one of the many other languages specialising in this domain). Erlang [Barklund and Virding, 1999] integrated network primitives directly into the language which gave some powerful expressive forms. The advantage is that the filter family can build on the knowledge that this information is intended for its consumption. A parallel or network operation using this non-native syntax can be converted into a form that suits the goal of the system and syntax of the underlying language even without the application being aware of how the operation will be translated.

The Shards system is capable of modifying application or system code just as it is capable

of modifying any structured data. Code can be simply another component to be generated as output before normal compilation of either system or application proceeds. All that is required are filter families that know how to recognise the input information and either use, consume or translate them before writing out the source code. It can do this because filters are passive and operate in sequence. The many filters that do not touch source code will make no changes or react to syntactic extension. The filters that understand the language in which the application load is expressed will perform analysis. The filters that understand hints from the application designer will analyse, remove or translate as appropriate. Finally, a filter will output a new version of any source code that has been modified. This can then be used to construct the application.

This level of interaction is fairly extreme as connecting the application software so directly with the Shards process is not a common necessity and creates a reduction in portability as the application now contains Shards specific content. There are also concerns about how useful application level input is to system design, though this concern is often framed within the general operating system context. The design of the Shards system provides this option but makes it low impact for when compatibility with a Shards free development environment is desirable.

This mechanism for application load code to be modified allows the Shards system to modify the source code even without direct instruction from the application author. This would be based on the Shards process performing analysis and being aware of specific system elements that could not be known to the application author. For example, the system designer provided metadata may identify the system as being distributed or parallel in some interesting way. A filter may know this configuration provides advantage or disadvantage to certain operations based on their structure. It could use this knowledge to modify the application load to take better advantage of the capabilities of the system. This could be implemented either using the target language or even directly compiling that code element to machine specific object code.

At this point there is information coming into the Shards system from the system designer, the application designer directly, through the assembled application load and from the filter authors. These combine to generate both system components and potentially modified applications. Both are designed to perform in an optimal manner relative to the goal of the software system and the platform in which it executes. It effectively makes system design a captured dialog between all the elements that contribute to an optimal system.

5.8 System Generation

The Shards process generates components to be integrated into the construction of the system. It can also be used to generate a complete system. It is expected that at the core of the system there will be a component that is not generated by the Shards system. This core will exist purely as a run-time engine to provide services for the other components that have been selected. One of the reasons it resists expression as a set of subcomponents is because it must be tightly integrated, focus on efficiency above clarity and can reasonably expect to be reused only as a complete unit (though it may be configured by the output of the Shards process). It is also expected that this core will be more distant and thus less directly influenced by the structure of the application load and other concerns at that level.

The core is intended to provide only the most primitive of operations. It is desirable for this core to be as limited as possible so that functionality is primarily contained in selectable filter family generated components. It is most likely to cover the same breadth as existing microkernel solutions but will probably be compiled into a monolithic executable for maximum run-time efficiency.

The boundary between the Shards components and this inner core can be considered the *core API*. This is where required functionality cannot be provided by generated components and thus the core must be called to provide service. It is desirable to have multiple cores that express different design strengths and approaches. These can become interchangeable by the system designer to the extent the core is universal. In practice switching cores may need some shim filters to assist in the translation of calls where differences exist.

The idea of a core API allows flexibility in the system constructed by the Shards process. The components generated by Shards could be used as part of an application or a layer integrated within some other system as long as calls on the core API are serviced. This would also allow components generated by the Shards system to be integrated into a microkernel system environment. The Shards system simply generates components and seeks to make minimal assumptions about what is built with those components.

It is also possible for there to be no core to the generated system. If the filters do not require functionality from the core then the core need not exist. This aspect allows for some interesting optimisation possibilities in terms of defining what the required core operations are. For example, consider a system in which only a single application will ever run and that process

will run constantly. Something that may be true for an embedded controller in an elevator or a mars probe. Such a system can disable or remove core functionality related to multiprocessing, memory protection, and other traditional core functionality it doesn't need. This may provide advantages beneficial to the design goal of the system. There is no modern general operating system that will accept this relegation to a non-existence, but the Shards process allows and encourages the flexibility required to enable this potential.

The ability of the Shards system to modify source code in the application load also allows for another optimisation possibility. Core functionality may instead be pushed into the application code. As an example consider a system that will always run two processes and switches between them. It can no longer discard multiprocessing functionality as there are two applications running. However it only needs an extremely simple structure since it always knows that a context switch will be to the other application. This allows a simpler context switch mechanic to be implemented and possible cooperation in terms of what state is left in place for the other process. This could be implemented by having the code to switch between processes built into the two applications. If the context switch is changed in a call on a Shards system provided library, and compiled into the processes, there is no need for core functionality for multiprocessing to exist. In effect the line between application and system functionality becomes permeable based on the needs of the system being constructed.

5.8.1 Shards Process Completion

It is expected that the focus of initial filters in the Shards process will be on performing analysis and generating metadata based on what is discovered. This will move towards filters which consume this metadata and generate actions based on their programming. The goal is to modify and select components, support libraries for those components and possibly applications that can be integrated into the construction of the final system.

As with all construction processes it is possible for bugs to exist and it is desirable to detect and repair them. Some of the information on the data chain can be regarded as requiring resolution. It is an indication by an analysis or action filter that a later filter must continue the process of generating the required changes or output. It is good practice that this data is removed or transformed when the filter performs the requested action. As such, it is possible to build late order filters that scan the data chain for unresolved tokens that should have been processed.

These filters can halt the Shards process and indicate that the filter order has logical gaps in the sequence that was followed. The data chain provides a list of the filters which have been processed, and which remain scheduled, allowing the process to be debugged. The element on the data chain that identifies the fault will also be a member of a filter family, allowing attention to be focused. The implementation also experimented with producing a Shards debugger which would record the changes in the data chain with each filter execution and allow the ability to step back through them.

The information on the data chain can otherwise be safely discarded on completion of the process. It is normal operation for metadata to be generated but not consumed. This is because early filters will try to provide as much information and as many optimisation opportunities as possible so that later filters have the widest range of data to draw from. This means they do not have to consider which filters have been scheduled to run by the system designer or may be scheduled to run by a later filter family. It is easier to generate data that is not used, even though this increases the memory footprint of the Shards process, than try to calculate precisely what information will be accessed and used.

5.9 Enabling Modularisation

The mechanisms of the Shards system are focused on allowing the very wide variety of possible system functionality to exist within a modular structure. This allows complex functionality to be contained within and manipulated as a Shards filter family. As much as possible the complexity is contained in the operation of the filter and they can be used without this complexity being examined or fully understood. This makes them more accessible than having to manually analyse and extract source code from existing systems before reuse is possible. If successful, this will make reuse of filter families more productive than manual creation or duplication of raw system resources. This also allows iterative development within the filter family to continue without disrupting systems using that functionality[1]

Constructing a reusable module is a challenging task that requires a complete understanding of the functionality contained within that component and how it relates to the demands and expectations of the external system. Constructing even a single component is a worthy project

[1]Naturally this is as long as the interface and dependencies do not change. The Shards implementation will support filter versioning for when they must.

for one or more skilled practitioners. The Shards system provides a structure into which it can be placed but does not stop the creation of a new filter family being a substantial amount of work. The practitioner may be able to gain from reuse of existing filters as subcomponents and existing filter families to provide other components needed to generate a full system.

Shards provides a solution to some of the issues that attempts to modularise operating system code have demonstrated. A module that seeks to be context aware will have to contain many possible variants to capitalise on all the possibilities. This makes the module larger, more complex and encourages it to develop complex run-time behaviours. The Shards system allows a module to examine its environment and select and tailor a matching implementation in order to potentially resolve run-time concerns in advance. This solution is then compiled into the system structure allowing a high level of run-time performance. This allows the filter family to be much simpler internally since it can be larger, perhaps including large sections of potential solutions as optimised code, knowing it does not need to be concerned with its own run-time efficiency.

The interface to a component is also problematic. There is a natural conflict between simplicity of interface so that reuse is eased and broad applicability of the interface so that it works in the widest number of environments. The best performance can be gained from an interface that is tightly bound to a specific system context which works against both simplicity and broad applicability. The Shards system allows some elements of this interface to be constructed during the operation of the Shards process. The filter family can have a very broad interface and automatically optimise around the calls actually seen in the data chain or the cooperative filter families it recognises. Since this is automated, it does not cause complexity[2] in the interface the system designer must understand to make use of the filter family. Since the resulting interface can be dynamically tailored, it can be both broad and optimised to the system context.

The final advantage is that the component is able to more actively draw on information about its context. Writing an efficient system component will involve a great number of design trade offs. Attempting to be ideal in all environments through the use of heuristics with limited access to system information (or a run-time cost in gathering it) can add a lot of complexity to the module. If the module can simply examine an environment variable, or calculate some element of the system environment or application, it can make stronger observations about the requirements. This certainty allows for simpler decision making and more effective determina-

[2]No such promises are made in regards to the process of debugging

tion of how to reflect this in the operation of the component. Analysis is enhanced since this determination happens before construction where there are no concerns with run-time efficiency. It is also possible for the filter family to replace complex analysis, processing and heuristics with an environment variable that the system designer can set in order to inform the determination of how to correctly optimise.

In practice, many components are generated via examining a working operating system and using its internal structure to indicate the line of division between potential components. The analysis required is very similar to that of someone learning the internal architecture of the system, whether as a student, developer or operating system implementer. The most likely source material would be one of the multiple Unix systems which are well understood in the academic environment, provide available source code and have relatively lenient licensing requirements. Unix also continues to be a viable and competitively practical operating system, which means its internal design is proven through use. This means that many models of operating system structure, which are the raw material for a component system, are directly or indirectly based on Unix (For example OSKit [Ford et al., 1997]).

This can be a disadvantage when trying to capture all system possibilities. It is a great simplification to use the structure of an existing and well known system as a foundation. The risk is that it is hard to avoid also bringing in a great many assumptions which limit what can be imagined or implemented. The Shards system focuses on making no assumptions about what the output system will look like or that other components will use a Unix-like model. The Shards approach encourages any assumptions about the system context to be made explicit. An assumption about the system will need to be linked with an environment variable or analysis process that then becomes visible and open to control by the system designer or module authors.

The Shards system is designed to be a foundation for both gathering modularised system components and building optimised systems from them. It is designed so that it can initially make small decisions about a single component in a system construction effort and scale up iteratively. It both informs and gains from system design as a discipline and provides a structure which can exist at both the theoretical and development level. In time it offers the potential for a rich pool of filter families which can allow faster and cheaper construction of customised systems. These will also provide opportunities for the evolution of new specialised mechanisms in the systems field.

Chapter 6

Related Work

The previous chapter introduced the goals of the Shards system. In broad terms there are three separate goals, none of which are connected to the construction of a specific implementation. One goal is to capture the design logic behind the construction of an operating system in a form that can be understood, communicated and manipulated. The second goal is to design a framework of structural components so that the implementation details can be encapsulated in order to avoid polluting the design decisions. The final goal is to have a process where the construction of the operating system implementation can be partially or fully automated in response to an input that represents the target workload of the finished system.

No direct equivalent to the Shards system was found in the literature. There are systems that have considered the same issue of operating system flexibility and proposed quite different solutions. This chapter will examine some of these systems that are close to Shards in their intent. The systems selected are not intended to represent a broad survey or the most recent developments in the operating system field.

The nature of documented systems work, which was discussed at the start of this thesis, also made finding equivalence more challenging. Since systems work is seen as tightly connected to its context there are relatively few general surveys or comparative examinations. This means that finding one system of interest does not necessarily lead to finding others. In addition many interesting operating systems become concealed by the more high profile application or context they support, which meant that at times system details had to be gleaned from other documents in which they were not actually the primary focus. The end result is that this section can make no realistic claim to be complete or without errors. With so many existing systems, and so few

established guides, some will be missed.

A survey of the contemporary field suggests the enthusiasm for novel operating systems has cooled. This is expected as the sophistication of the mature systems increases and producing a competitive equivalent becomes a larger project. The value of a rich software ecosystem has been recognised which also acts to discourage the emergence of new operating systems. The Linux system has been extremely successful and now provides both a foundation for new system domains such as smartphones and laptops and research into operating systems algorithms and mechanisms. A quote from the K42 project [Krieger et al., 2006], which was funded by IBM and staffed with experienced domain experts, provides a expert reflection on the environment for operating system research.

> There has been a marked decline in the number of complete operating-system initiatives. The environment for such initiatives is no longer conducive for several reasons. Pressures from academic publication volume or industrial research deliverables have increased. More importantly, the number of legacy interfaces and applications that must be supported for a system to be relevant has increased with the growing user base. In the ten years before we began K42, a large number of complete OS projects were undertaken, including Chorus, MACH, Sprite, Synthesis, Peace, Amoeba, Clouds, Spring, Apertos, Choices, Opal, VINO, Plan9, Exokernel, SPIN, Rialto, Paramecium, Nemesis, Scout, Tornado, Eros. In the last ten years, fewer complete OS projects have been started. Only K42 and Flux , and more recently Singularity and Asbestos are examples. We do not imply that OS research does not occur, rather that research into whole or complete operating systems has declined.
>
> - *K42: lessons for the OS community* [Wisniewski et al., 2008]

6.1 Minimal and Extensible Machines

The Shards system was partly inspired by the concept of automata, which are logical constructs that offer a combination of mechanism and definition in a concise package. The scale of the Shards system means that practical questions relating to complexity and ease of application have to be considered. In addition a great deal of the literature in automata theory is primarily interested in the theoretical aspects that these machines reveal in their operation, or for

constructing logical proofs. They can also be considered models for the essential operations of a computing platform and as such are useful when thinking about constructing a flexible framework for system construction like Shards.

One early idea was the use of a simplified virtual CPU as a logical foundation. This, combined with a form of assembler adapted to it, would provide a rigorous combination of mechanism and definition for testing operating system component models. The complexity of the solution could be reduced by using procedural structures such that complicated sequences of code could be concealed behind a regular interface. This is somewhat similar to the approach used by Knuth in his MMIX language [Knuth, 2000] which he used for both proving and demonstrating the algorithms contained in his text.

There are limitations with such a model however. The first is complexity and redundancy. In order to encompass the scale being considered higher level language constructs would be required. However the higher level the structures the less concise a definition an implementation provides and the more inflexible language structures are required. In addition it can be seen that the definition and the actual implementation are mirroring each other if we are expressing theory in such a fashion. If this is the case then it is easier to simply consider the implementation, once written, as the definition.

There are also some passive and static behaviours inherent in this model. The core CPU is constant in its operations, but since there is probably little interest in redefining such primitive operations it is not a substantial problem. However functions once written, and their invocation by name, are not flexible in such a model. They can be made flexible using by using self-modifying code[1] and call tables, but this leads to a substantial increase in complexity. Even if this is done, the code rewritten and the calling name altered, it will still only apply to code that actively calls it. The machinery can not be universally reprogrammed which limits how dynamically expressive the system can be[2].

[1]Self modifying code is a recognition that code exists in the computer's memory and thus a program can rewrite itself. This is non-trivial given that the structure within the code is likely to be complex and most hardware and systems will assume it is static during execution.

[2]This assertion is not true within the domain of Field Programmable Gate Arrays (FPGA) which are in fact dynamically re-programmable [Saleeba, 1998]. However the size and speed limits of the hardware means they are not a substitute for a standard CPU for computing tasks so the crossover is limited. Similarly when operating as flexible hardware they are unlikely to be using their available gates to support operating system functionality. If such a system did exist however this capacity could be integrated into the Shards system either as a target or as a module which included the facility to reprogram the FPGA and link its functionality into the wider operating system.

This is partly a result of compilation and execution being two separate phases in most procedural languages. This automatically tends towards functions being fully specified as constants before execution begins. There are dynamic languages that are more flexible in that regard. Lisp [Steele, Jr., 1990], which uses lists as a data structure can declare functions at run-time. Forth [Bishop, 1984], which is stack based, can do the same via a function dictionary. However it remains true that these functions must be explicitly called by the code. The execution engine itself is not adaptable at run time, partly in the interests of practical efficiency.

A much simpler mechanism that is entirely flexible can be found in the model of a Turing Machine [Turing, 1937]. In this system the core mechanism is the recognition of encoded operations in the data stream which is then checked against a table to find instructions on how to progress (which can include changing the data). In this design if the content of the table is changed, which could occur due to the actions of software, then every element of the program may respond differently. It is also possible that a section of data, which might be the product of an ongoing computation, could be declared to be an encoded line which should be placed in the table.

The only operations that a Turing machine reserves are extremely primitive and neutral output operations directly related to the operation of the machine. Two instructions involve moving the read head through the data, which is visualised as a one dimensional tape. Two others involve reading and writing symbols from the tape. This is sufficient mechanism to enable the expression of all computable functions. It is true that the concise architecture is reflected in a far from concise functional expression however.

It can be seen that the Shards system owes a great deal to this model, although it has added much complexity in the interests of practical application. The paper tape of the Turing machine is related to the Shards data chain. While there is not an unlimited ability to reverse the tape (in the interests of parallelisation) the ability to rewind uncommitted data and re-process the entire chain through scheduling the same filter again, has the same net effect. Each filter represents an independent translation table, which like the Turing machine responds to its internal state and the data currently being read. Scheduling the next filter represents run-time control of the engine's nature. The decision whether to pass data unchanged or modified representing the Turing machine's ability to write to the tape.

The main theoretical difference is that the Turing machine was intended to be a purely

mental construct for the proving of some specialised mathematical concerns. The Shards system inherits some of the mechanism but the goal is a practical machine for the construction of complex structures. This results in significantly more state being held, the data channel widened and the operators being extended. Whereas the Turing machine needs only four operators, the Shards system makes all the operators within the chosen filter implementation language available for processing. This degradation of the model allows far more concise and convenient expression of operations.

6.2 Compliant Systems Architecture

One project that has a similar aim to the Shards system is the Compliant Systems Architecture (CSA) project. This project has a core paper [Morrison and Balasubramaniam, 2000] which aims for substantial gains by allowing tighter communication between application and execution. A primary method is the provision of system-wide policy which expresses the direct connection between these two parts. It can be seen that this is another expression of the difference between general and custom systems. All custom (non-general) operating systems, of which the CSA foundation system is one, will measure their success by the extent they allow a clear and efficient connection between application and execution environment.

The CSA project is actually a global term for a system comprised from a number of semi-independent subsystems that have evolved to work together. These lines of development have a substantial heritage in their own right. The CSA papers do not always make this system environment clear, or the connections explicit. Observations derived from the custom environment of the base system are announced as general observations globally applicable to the systems domain.

The most important element for understanding the sequence of development is the IPSE 2.5 project [Warboys, 1990] which is a high level software engineering environment. It uses a technique called process modelling which defines a large software project, in this case ICL's VME operating system, as a massive data structure. This structure encapsulates code, developer roles and tools within a single structure. The modern variant used in the CSA project is known as 'ProcessBase'. This continuously evolving structure, of a substantial scale, has a natural correlation with the concept of persistent operating systems [Dearle et al., 1994] which are well suited to hosting such an environment. The low level requirements of the system are met by the

Arena [Mayes and Bridgland, 1997; Mayes, 2001] project which provides basic operating system support.

Once considered as a specific system the drivers behind development become quite clear. The system is built to support a specific software architecture that is extremely demanding in terms of processor and memory use. This demanding process will be relatively slow, which is acceptable since it is focused on supporting human operations, specifically software construction. More precisely the model given calls for six layers of largely independent but cooperating software. These include a microkernel (called the Hardware object or HWO), user space system libraries (called Arena Resource Managers or ARMs), a virtual machine (called the Process Base Arena manager or PBAM), the dynamic compiler (Process Base itself), the resident functions and finally the resident process model. Many of these components are dynamic and independent systems which contain some elements that are traditionally system functionality. For example the persistent systems component wants control of memory mapping and protection schemes, something even most microkernels do not allow to the extent this project desires. Also because they are dynamic and autonomous systems they cannot be integrated for run-time efficiency, even if that was a primary concern.

This combines well with the interests of the Arena system itself which is basically a low level microkernel with a collection of replaceable user space libraries. This is not particularly novel, being typical of microkernel construction. The unusual element is how little is contained within the core of this system. Many microkernels, for efficiency reasons, support processes and basic memory mapping within the kernel. Arena does not, willingly passing control to potentially application defined user space libraries. This does mean that the Arena system needs to frequently pass system events to the user space libraries, called 'upcalls', so that they can be handled. This is convenient to both persistent systems and the process base model which want substantial control of relatively low level operations. The papers are reasonably silent on the overall performance of the system, although it is hinted [Mayes, 2001] that some concessions had to be made in the interests of efficiency.

The end result is a system which is inherently constructed from layers that are independently active at run-time. These layers also implement low level functionality which would normally exist in the core of the microkernel. Since many of these layers are dynamic, or even interpreters, it is also possible to define flexible interaction patterns at run-time. It should be clear that

already this is a relatively specialised environment. If the goal was primarily performance driven, which is a common element of custom operating systems, then this architecture is inherently sub-optimal but is required to provide support and functionality to the higher level layers of the system which are the focus. This is one of the reasons why system optimisations and observations cannot be easily abstracted from their system of origin.

The primary theoretical contribution is contained in the formulation of policy and mechanism, which are described as separate entities within the system. Specifically the policy at any given level shapes the mechanisms provided by the underlying layers. The combination provides the mechanism for the next software level. The policy is compliant in that at each level it meets the best needs of the application. Broadly speaking this is not a novel observation. All operating systems can be modelled in this way as a software layer selecting which underlying functionality they expose and which they encapsulate or abstract. All systems also measure their success in how well they meet the demands of the application software, as far as those demands are known. Thus it is somewhat difficult to visualise a non-compliant system given the description presented [Morrison and Balasubramaniam, 2000].

The actual compliance element is certainly well observed, there are advantages to the system being shaped by the application and avoiding duplication of functionality, something achievable in the CSA example because the system has been designed to support a single application. However the demands of the application under consideration, and the existence of multiple independent run-time configurable layers allowing flexible definition of policy, are atypical. In addition generally applicable heuristics for generating these interactions are unclear. The creation of a CSA will still require a significant creative act by the programmer, although the separation between systems and application programmer has dissolved.

The main difference in approach is that the CSA method is focused on a particular kind of application, which is a persistent process with a very large memory footprint. Shards is designed to make no assumptions about the software architectures it may be asked to construct. In addition the focus of the CSA system is optimisation over the entirety of the system. Since the solutions are system wide and because the system is quite heavy-weight and long lived, it is acceptable to represent the different architecture levels as structures in memory and perform their tailoring at run time. The Shards system aims to integrate and tailor fundamental operations of a much simpler, lower level and efficiency constrained system. The Shards system also aims

to resolve issues at build time wherever possible so there is no run time penalty either in size or time. The difference in focus between the two projects means there is little direct cross-over in either direction.

6.3 Code Generation

A line of operating system development that sounds, at first, similar to this project is that focused on code generation. After all, the ultimate aim of this project is the generation of system code in response to discovered application need. Deeper investigation reveals significant differences however, primarily in the scope and target to which code generation is applied.

This branch of operating system development springs from the Synthesis kernel [Massalin, 1992]. This kernel, written in 68030 assembler for the NEWS workstation, implements a number of interesting operating system properties in search of greater performance. One of these techniques is run-time code generation. The essence of this technique is that at run-time certain parameters, and thus control paths, become known. If those parameters are invariant, during the life of the task or operation, then it is possible to generate code that implicitly includes this state knowledge. In other words efficiency can be realised by devolving the code from a general case, common in operating systems, to the specific case needed in a given instance.

The difference is that the code generated is not meaningful at the architectural level. The path through the system is simplified, through the removal of redundant operations given the specific state information, but not changed. Stated another way it emphasises the path that would have been taken, but does not extend to considering the possibility of there being different ways in which the path could be structured. This is partly a result of the fact that it is being performed at run-time, which means that excessive analysis work in compilation might consume more time than the optimisation ends up gaining. So while it is a useful optimisation technique, and valuable in allowing the system to offer many possible paths without a performance overhead for unused tasks it is run-time and implementation focused whereas Shards focuses on system construction. The two approaches could be complimentary but do not significantly overlap.

The line of development continued in the Synthetix project [Pu et al., 1995] which includes the concept of 'quasi-invariant' variables. These are variables that are expected to be invariant, but that the system must be able to recognise and cope with this assumption proving incorrect. The Scout project also uses code generation, although not at run-time, in the Filter-

Fusion [Proebsting and Watterson, 1996] compiler. This compiler is able to merge filters, written as a separate entities for architectural clarity, into a single software construct. It does this by following the complete range of data paths through the various filters, a process that can be expected to be compatible with run-time specialisation as state becomes known. Once again it makes no impact on the systems architectural design since it focuses on optimising the existing paths the software contains.

6.4 Minimal Kernels

One approach to flexibility is to remove "required assumptions" from the system architecture. In other words if the operating system does not specify behaviour then there is inherently greater flexibility in the overall system. The logic is that this absence at the system level will be replaced with a much more application sensitive implementation at a higher software level. The challenge is thus seen as exploring how few assumptions are needed to make a functionally useful system. This is motivated by the observation that microkernel systems, specifically Mach [Rashid et al., 1989], had grown so large and tightly inter-connected that they were in practice no more flexible than more traditional systems. The result would ideally also serve as a model to identify the truly fundamental operating system mechanisms that can not be delegated to an external agent.

One of the most referenced projects is the Exokernel [Engler, 1999a; Engler and Kaashoek, 1995]. The essence of the Exokernel is in re-drawing the line of demarcation between system and application to the lowest possible point. It aims to produce an operating system which hides nothing, understands nothing, and makes extremely few assumptions, which is actually an extremely challenging technical task. It does this by directly exposing the hardware interface itself so that virtually all functionality must be contained within the application, presumably through the inclusion of predefined library packages. At the same time it allows this access in a controlled manner. Applications must ask for ownership of hardware resources, will have that resource protected and can even have it revoked if required for the greater good of the system.

It uses some extremely clever mechanisms to make this possible and efficient. One example is that it is able to protect disk allocations even though it has no idea of the allocation mechanism being used. It does this by effectively requiring the application to provide it with the tools it needs, in a form it can trust, so that it can check if the access is legal. Another example comes in the form of incoming network packets. This is a more interesting example because it does not

know the format and as a result cannot identify the owner to provide assistance. The answer is that the applications provide code, with some specific qualities including the structure of the language used, that the Exokernel will include in itself.

There are however two concerns. The first is the technical concern that widespread applicability has not been shown. The system has been demonstrated to work in the hands of the creators, but not whether an external developer with less understanding finds it easier to work with than the alternatives (such as Mach [Rashid et al., 1989]). The durability of their mechanisms, against multiple applications with competing demands and varied operation, would also be extremely interesting. It seems it is safe to suggest that the existence of openings for malicious interference within the Exokernel system cannot be ruled out on current evidence.

The more serious concern, however, is architectural. The Exokernel has successfully moved complexity from the kernel into user space libraries. It has achieved this feat at a cost in complexity to itself, since it must implement multiple mechanisms to make it possible, but also in the application and library layer which must deal virtually directly with the raw hardware. It is unclear whether this trade off is broadly beneficial as the architecture is reasonably complicated and many applications may not be able to substantially benefit from the difference.

The system offers very few clues or tools for constructing the user space libraries that will end up doing much of the work. They are still constructed by hand in response to the needs of a given context. They are still likely to depend on tight interaction with each other, both in mechanism and in conceptual model, if the system is to be efficient overall. It is still likely that the existing libraries (for example the Exokernel developers wrote Unix-like layers above their kernel) will be reused wherever possible even when not truly optimal because this reuse represents a massive reduction in complexity. In short while the project has optimised the flexibility of the kernel it has not solved, or even necessarily substantially reduced, the complexity of writing a complete system tailored to the applications needs.

The Exokernel project has some interesting elements. For example it makes use of the compiler as a trusted component of the system. Specifically the code that is down-loaded into the Exokernel is expressed in a restricted language which makes exploiting the Exokernel's inherent weaknesses much harder. A subsidiary paper within the project [Engler, 1999b] uses the compiler in another fashion, one that is concerned with improving the binding between the programming language and a target interface.

The Exokernel is also interesting as an argument about what constitutes a minimal operating system. It effectively states that securing access to the machine resources is the only system role that could not be entirely moved out of the system core. The Exokernel is however based on the assumption of one kernel, minimal though it may be, existing across all applications. In other words it remains a general core that does not shape itself to specific demands of a given system. Since there are systems that answer the question of what constitutes a minimal kernel in different ways then the Exokernel model can not be assumed to be optimal in all cases. A very simple example would be to postulate a system which will only ever run a single application. In such a model verifying its permission to access hardware is not a concern and the mechanisms of the Exokernel are sub-optimal.

The Pebble Component-Based Operating System [Gabber et al., 1999; Magoutis et al., 2000] is an example of an attempt to construct the absolute minimum core while maintaining efficient performance. This kernel focuses purely on the process of transition between functional components each kept in their own protected memory space. The central mechanism is a *portal* which is effectively a connection between two components within the system. For the caller it is a look up table connecting it to a body of code in another protection domain. This code is dynamically generated from an interface specification in the server's protection domain. The constructed interface between the two also manages various elements of memory mapping which allows the two components to potentially share memory pages for communication. This allows transitions between protected domains to be accomplished with minimal work or capability on the part of the operating system core.

The Shards system could comfortably use these minimal kernels as the OS core for the constructed system. These mechanisms are clever and provably efficient, as are well respected microkernels like L4 [Haeberlen et al., 2001] and provide valuable system primitives. There is less attention given to the structure and functionality of the user-space servers which will ultimately contain the majority of the complexity. The design logic behind these servers tends to be driven by the actual mechanism, rather than specific implementation requirements. It is this complementary part of the solution on which the Shards systems will focus.

6.4.1 Microkernel Variants

The microkernel domain has been one of the dominant fields for operating system innovation. This is unsurprising because the architectural design of a microkernel is suited towards evolutionary adaptation by small groups. However the microkernel, because it maintains this architectural model at run time, has also proven to have a performance cost. This has been a negative for commercial application, but it has proven to be another driver for system development.

The primary weakness of the microkernel approach is that communication between the microkernel and the operating system components is frequent and not free. However this cost is multiplied when the idea of a protected kernel memory space, which is beneficial for system stability, is introduced. Now each communication absorbs the cost of a hardware assisted, expensive, mode switch. *Collocation*, the technique of moving services back into kernel space, can reduce this cost but at a reduction in security, stability and the architectural model of the microkernel. The following three systems are examples of specialised responses to this challenge.

The SPIN [Bershad et al., 1995b] kernel is perhaps the most general. It responds to the communication cost between the microkernel and system services by moving functional interfaces into kernel space. This also allows the microkernel to be shrunk even further as there is now no double handling of responsibility. Instead the micro kernel becomes simply a router of functions and events to the service. The security risk inherent in this collocation is solved through the use of restricted compiling. Forcing the use of a pointer safe language in a trusted compiler means that any failure will be contained with the service that caused it. The central idea, of using a compiler in order to generated trusted components, is one that is central to the Shards system. However the Cache kernel, which we will examine next, mentions that kernel support of the resources needed by these functions has some complexity.

The Cache [Cheriton and Duda, 1994a] kernel keeps the functional elements in user-space. It avoids the performance penalty for this by identifying that functional elements are frequently represented by data objects which hold their state. Much of the communication between microkernel and user-space system service actually represents access to these data structures. The Cache kernel optimises this by storing (caching in its terms) the data structure in kernel memory. This generally means that application requests can be directly satisfied by the kernel using the stored data structures. Only when there is a change in system state must it communicate with the system services to update the data structures. Since data access and usage are far

more common than changes in the system state performance is improved. And unlike absorbing functional elements, data, being passive, is much safer and simpler to support. The solution is made somewhat less general because the operating system is melded to a hardware system development effort. There is also some complexity in the dependency between data structures.

The Vino [Small and Seltzer, 1994] kernel is another example that chooses to specialise the interfaces. Effectively it argues that the interaction between kernel and application is not only slow but its generality makes it poorly suited for many specific purposes, in this case database systems. Its answer is to define basic operating system interfaces at compile time but allow drivers to be loaded into the kernel at run time that replace the default behaviour. It refers to this as grafting. Like SPIN the compiler must be trusted to ensure that these grafts can be trusted, but because their function is much narrower they place fewer demands on the system than the SPIN equivalent.

The three systems are introduced because they offer three different solutions. In essence SPIN allows functional elements to become part of the kernel. The Cache system allows data structures to become part of the kernel. The Vino system allows interfaces to be both specialised and controlled from within the kernel. They serve primarily as demonstrations that the definition of what functions are within a system are negotiable, and there are strengths and weaknesses to each decision. There are many more microkernel variations, many of a more evolutionary nature, sufficient to easily fill up a chapter or even a thesis by themselves. However it can also be noted that most of them cannot solve the central problem, which is that the architectural advantages of the microkernel model come at a run-time performance cost, and the notion of a minimal operating system is often context dependent.

6.4.2 General Properties

In looking at multiple microkernel systems, more than are represented here, some interesting generalities were observed. The first is that a great many of the systems, those that provide comparative benchmarks at all, compare themselves to OSF/1 and Mach [Rashid et al., 1989]. This is interesting because OSF/1 is itself a Mach based system, making the test less valuable than it may appear. It has also been established that the Mach system, as one of the originators of the microkernel architecture, is relatively heavy weight and sluggish. Comparisons against more modern microkernels, or contemporary monolithic Unix kernels, would have been more

useful. The SPIN [Bershad et al., 1995b] is one example of this.

It was also noted that for many of the general purpose systems the first application written was a Unix emulator. Mach, SPIN and the Cache kernel all have Unix emulators. This is a natural progression in that it gives the emerging platform access to an established software base. However it also became obvious that few of these environments managed to generate customised software that emphasised their individual strengths. It is reasonable to suspect that the existence of emulated versions brought convenience equal to the extent with which it suppressed the development of local equivalents. This could be called 'suffocation by emulation' and it is a widespread phenomenon amongst experimental operating systems. It is more visible on microkernels however as their design assumes that many systems services are externally provided.

6.5 OSkit

One of the projects with the greatest affinity with, and influence on, this project was the OSkit [Ford et al., 1997] development effort. This project begins with a similar premise, that the effort involved in constructing a minimum workable system makes research in the systems domain impractical. They also observe that while there are multiple source available operating systems each tends to be tightly coupled with limited modularisation, cross kernel function calls in the interest of efficiency and many shared but unstated design assumptions. This makes separating an existing kernel into reusable chunks complicated. It also means that trying to integrate a novel component will be difficult unless the component follows the architectural expectations of the host system.

The interest in support for operating systems research was a result of the Fluke [Ford et al., 1996] operating system development project. This project aimed to express a system in the form of layered virtual machines. This means that the base operating system, if any, is defined by the functionality that is not contained in the higher level layers. This relationship means that the underlying operating system is regarded as a provider of services, it cannot assume permanent possession of any system facility. Even system facilities considered as existing only within the operating systems domain, such as process management, could be partly or fully contained in higher level software systems. The end result is that the fluke project wanted full control of the structure and extent of the underlying operating system, but wasn't actually interested in

writing it.

The approach used in the OSkit can be summarised as relying on modularisation of system structures. Even monolithic operating systems, such as BSD and Linux, have an internal structure. This is often quite regular as some system facilities, for example device drivers, exist in large numbers and are constructed using interfaces internal to the kernel to reduce complexity for kernel and driver developers. The OSkit exploits this internal structure to break the existing operating system into reasonably clean modules, although the granularity tends to be reasonably large. This large granularity is also advantageous because the system intends to directly reuse existing code to avoid the effort of creating and maintaining local equivalents of functionality already existing in other systems.

The OSkit is profoundly practical in its approach. Its use of existing kernel structure reduces, but does not remove, the complexity involved in establishing modularity. This modularity can also be supported in the interface between modules, through using function wrappers and call re-direction, which means no changes need to occur in the body of the code itself. These two mechanisms enable the OSkit project to reach a practically useful state much faster than a system constructed from scratch can hope to match. The paper [Ford et al., 1997] states that the OSkit absorbed 230,000 lines of code from existing systems, which represents a substantial saving in development time.

The limitations of the OSkit primarily stem from this practical approach. The ability to directly reuse code means that the system structure, while modular, reflects the parent systems. A system that is incompatible with the Unix code the OSkit depends on can be assumed to be incompatible with the OSkit itself. Comprehending the structure of the many modules that make up the OSkit is also dependent on assumed knowledge of Unix structure. Without this higher level knowledge determining the application of the many independent parts of the OSkit would be complex. In short the designers have produced a modular Unix, rather than a truly universal module library.

This is not a fault or flaw in the OSkit. The goal of the designers was to produce a practical system whose elements could be controlled in detail. An attempt to support all possible permutations of how an operating system could be constructed would not improve this application. The system would need to be constructed from extremely primitive components so that no assumptions are built into the system. This would substantially reduce the ability of the

OSkit to leverage components and code functionality from existing operating systems which would increase complexity and the amount of effort required to release a practical system. Indeed any attempt to produce a truly universal version of something like OSkit will choke on the exponential complexity involved in not being able to make assumptions about structure.

Another limitation is that the OSkit must deal with the fact that existing code does not cleanly segment. There are system-wide constructs in kernels, for example the internal memory buffers by which elements in kernels communicate, which all subsystems within the system use and make assumptions about. A significant amount of the paper [Ford et al., 1997] discussed dealing with these issues. The OSkit, because it is dependent on foreign code, is unable to remove this dependency. Instead it relies on intermediary software layers (called 'glue') to emulate the native environment. The result is that the OSkit is forced to dynamically resolve incompatibilities at run-time which enforces a performance penalty and means these incompatibilities are hidden behind the provided interfaces.

It is also worth mentioning that kernel performance optimisation tends to depend on the efficient use of cross-kernel structures. For example most Unix systems use memory buffers that are shared between all elements within the kernel. Optimising the structure and operation of this buffer system can offer performance gains precisely because it is widely used throughout the kernel. In the OSkit system, which is forced to enforce modularity at run-time, the optimisation can only be applied within the module being considered. Not only is this likely to lose the performance benefit it may reduce performance through complicating the emulation the module must do. In simple terms the OSkit approach works best with simple kernels.

6.6 Object Oriented Operating Systems

There are a number of operating system implementations based on programming language mechanics. One of the more productive results involves using the object oriented paradigm to describe and componentise the operating system. An example of this is the PURE [Beuche et al., 1999] family of operating systems. The synergy is quite clear as operating systems are large bodies of code which ideally contain a strong component model and object oriented programming languages focus on making such component boundaries and interactions visible. Once the language has been specified as a large object model subcomponents can be modified or replaced to provide flexible adaptation. The compiler provides the capability to convert this model

into a unified and efficient executable form. This allows PURE to suggest a *family* of operating systems in which functional blocks are included in the system only if the context requires the functionality. It targets embedded applications where a general purpose operating system would frequently be providing excessive functionality in a resource constrained environment.

One concern with many of these projects is whether they add substantial new insight into the domain of operating systems. It is unarguable that modern software engineering techniques such as object oriented design allow large amounts of code to be structured in a more concise and efficient fashion. This increased internal structure naturally makes manipulation of the components within the design much easier. It is this that allows PURE to be more easily tailored to a specific target environment. A well defined internal structure is as much about good software design as the language used to construct it or the methodology used. No operating system aims to have a weak architectural model with confusing lines of division between functional areas. And some that do may have valid implementation optimisation reasons why the structure is less than ideal as a model of software architecture.

There are also limits to the flexibility an object oriented operating system can provide. As with OSkit a practical implementation still depends on having an internal structure that allows the component boundaries to be identified. The lines of delineation between the components are then encoded into their interfaces and thus become dependencies. This means that flexibility is restricted to the component level rather than structure or implementation. At the same time a great deal of the complexity in constructing an optimal operating system still exists within the objects themselves. This complexity may even be aggravated because of the need to work through narrowly defined and pre-existing interfaces. This can be seen in the PURE system which relies on a small number of core objects which means each object must encapsulate substantial internal complexity.

A variation on object oriented systems, but one that covers a similar territory, are operating systems based on dynamic interfaces. This is similar to object oriented operating systems in that they seek the construction efficiency of a modular architecture. It differs in that rather than using the object paradigm for defining their boundaries and interconnects they tend to use a run-time binding mechanism similar the CORBA mechanism [Group, 2006]. In theory this allows a degree of flexibility but it also comes at a run-time cost. Since the need to dynamically define bindings is relatively rare (and even in a CORBA based structure not trivial to design

into the model) while a run-time cost is considered undesirable it has had a mixed response. The THINK [Fassino et al., 2002] system is an example of this approach.

6.7 XML / SAX / XPath / XSLT

These technologies that are derived from the extensible markup language(XML) are not traditionally considered to be related to the operating systems field. They are now stable and have established formal standards [W3CXML:2004] though as standards these are focused on precise expression rather than readability which can make a guidebook [Harold and Means, 2002] invaluable. The similarity to the interests of Shards comes from the fact that XML is a flexible vehicle for carrying structured data without making assumptions in advance about the meaning or structure of this data. Since the data is structured other tools have come into existence to automate the manipulation of XML data. If this progressed towards expressing systems in XML and generating output it could offer some insight into the issues of data representation. The tools mentioned, XML, SAX, XPath, XSLT as well as other extension and user provided code are required to reach this level of capability which makes the resulting system quite complex to understand.

The first correlation is that the XML concept of tagged and structured data, while not novel, has similarities to the Shards data chain. Both of them contain data which includes tags identifying the semantic role that data plays. XML is significantly more verbose, being designed around static data stored in character files. It also includes the concept of syntax rules being contained in an external document known as a data type definition(DTD) for which Shards has no equivalent as it considers syntax to be defined by the manipulator and not as a pre-existing property. This level of similarity is not actually notable however, unlike the external data files that were the origin point for XML, programming languages have had the concept of named and typed data very early on in their development.

XML data has a relatively simple, but sufficiently strict, default syntax that makes it parsable. It was a logical development to automate the translation between source and output data using the concept of template expansion as found in the object oriented programming model. With an external framework, such as the Cocoon system [Langham and Ziegeler, 2002], it is even possible to script a sequence of transforms to be performed in a given order. This process, constructed from a variety of modules, is functionally similar to the Shards systems

sequential application of filters which modify the data chain.

The actual functionality in the Cocoon environment begins with the SAX (Simple API for XML) parser which reads the source XML and generates events corresponding to every data node encountered. These events trigger templates within an XSLT (Extension Style Sheet Transformations) file which output data, which may itself be a valid XML file and thus input for another cycle of parsing. XSLT also depends heavily on XPath to enable templates to specify which nodes they match. It is possible to replace SAX with DOM (Document Object Model) that parses the XML file into memory and then acts as an interface for access. This is not used in Cocoon because the expanded XML can be quite memory intensive.

At a theoretical level this approach and the Shards system are similar. Both provide a data format and the means to apply a sequence of transformations to it. The primary practical difference is that using XML in this way is an extended functionality in the XML environment, which is primarily concerned with data management, as against a core functionality in the Shards system. The Shards system is also not dependent on a tree structure (which contains an implicit assumption about syntax relationships), the relatively narrow match flexibility provided by XPath and the verbose structure required in order for both XML and XSLT itself to be machine parsable. Because the Shards system opens the filter sequence, and the match process, directly to the application code, it can be far more flexible than the XSLT equivalent.

For a project where the data structure is the primary interest, and the transformations direct and cleanly expressible, XML / SAX/ XSLT and a framework like Cocoon might be well suited. At this level the automated matching and transformation provided by XSLT and XPath reduces complexity because its basic operations are predefined and can be invoked directly. This also means that the range of operations is more predictable, especially since XSLT is extremely regular in format and restricted in operations. However advanced matching, sequencing and parsing, which will be forced into user coded extensions in an XML based system, degrade this advantage and begin to run into restrictions in the assumptions of the provided tools. As such these tasks, many that require substantial extensions coded in Java, are likely to be better suited to the Shards approach which assumes this level of flexibility will be required.

While there exist some informal sources [Sarkar and Cleveland, 2001] recognising the theoretical possibility of system construction using these tools there is no indication that such a product, with complexity equivalent to a compiler, has been constructed.

Chapter 7

Shards Design

The goal of the Shards system, as developed in chapter 5, is to capture and automate the process of system generation. This chapter focuses on the mechanics of how this could be implemented. The implementation should be concise in its operation so that it can be readily understood and extended. At the same time it should be flexible so that the implementation is applicable to the widest range of possible operating system construction efforts. This discussion will form a design document for the creation of the Shards code base, which will be expanded in the next chapter and its application demonstrated in chapter 9. The mechanisms presented are intended to provide a starting point for the development and automation of filters. It is expected that the path to a mature expression of the Shards concept will be extended and iterative for both the framework and the number of filters available.

Shards must rely on interested parties to implement, reuse and extend the functionality they require. These interested parties will be system developers or experimenters. They will be faced with the choice of writing custom code, modifying an existing system or possibly using a system generator like the Shards system. In order to grow the Shards system must be able to provide advantage through reuse or convenient automation to balance the extra complexity generalisation can add. This is especially true when the Shards system is itself immature and thus has limited capability to offer mature filter families for reuse. In other words the Shards system must convince a potential user that it will be able to offer advantage in achieving their own project goals or it will not be used. Automation is a core element of making this argument possible.

The other aspect of the approach is what is referred to as *fall-through*. The essence of the idea

is that the Shards system can be productive even with only partial coverage of the project. This is why Shards exists as a complement rather than as a replacement for the existing operating system generation methods. A Shards application could be as simple as performing some analysis which makes slight modifications to a source code file (or perhaps just the project makefile) ahead of the normal operating system build process. Even in a fully expanded Shards system it is expected there are many sections of the operating system that do not justify customisation and are compiled as normal. This ability allows Shards to be applied and expanded iteratively even within the context of a project.

7.1 Overview

A good tool should have a clear function if users are to understand the value it can offer them. This is made more challenging since the Shards system is effectively a platform rather than a complete solution. However the essential operation can be expressed relatively simply.

It is expected that full operating system development efforts will continue to require a software team. While Shards provides a structure for how the system can be expressed and additional potential for reuse it does not change the fact that operating systems will remain complex and substantial software engineering efforts. This allows a great deal of system knowledge and development capacity to be assumed. Actual end users of the system will run applications on the completed operating system and will not come into direct contact with the Shards system itself[1].

The first step is to construct a model of the operating system that can also be used to guide operation. This is a machine readable form so it does not replace abstract models in design documents and in practice should be derived from those models. It aims to capture the primary inputs that Shards can use in the process of construction. The document is called the *project file*. It is a document with a regular format that describes the explicit guidance of the operating system designer in terms of environment variables which are primary input for filters. For example a filter doing memory management optimisations may want to know whether the design goal is size or speed efficiency. This can not be derived from the source code as it is dependent on the design intent of the system. Instead, the filter documents that it recognises

[1]It is possible in some situations that end users might rebuild the system with a new application load as this part of the process could be fully automated.

certain values and will act on those values. In the absence of a defined value it will follow a general optimisation policy. An environment variable can also be a more general environment value defined by the Shards system on the expectation many filters will be interested in the value if it is set (for example the architecture of the target hardware). Indeed the question of whether to optimise for memory image size or speed is well suited to be a standard global value as many filters might be interested in this value.

The project file will also contain a list of filters organised in sequential order. Each of these filters is capable of taking arguments to tune its behaviour or indicate input it should operate on. The filters the designer puts into the project file are expected to be very high level filters, such as a filter that claims to do some modification to memory handling. As with all good software the filter will attempt to conceal complexity in its internal construction and operation where possible.

The final section of the project file is a list of source code files representing the selected application load. This section may be empty if the Shards operation is not dependent on analysis of client behaviour. The application load is expected to be a number of client applications that represent the tasks that will be used as a target for operating system optimisation. This may not be the entire body of software that will run on the system. It will not contain application code which is not valuable enough to guide the optimisation of the system. It also may omit application code which is known to have no interaction with the selected Shards filters. This source code may potentially be modified into a temporary file and compiled to object code by a Shards filter if part of its solution involves some re-writing of the client source code.

The Shards system uses traditional parsing techniques, and tools such as Lex and Yacc [Levine et al., 1992] to convert the project file and the indicated application load into a single easily manipulated data structure referred to as the *data chain*. It may even use existing compiler front ends such as LCC [Fraser and Hanson, 1995] or a language specific precompiler to convert source code into more usable internal data forms. This can be very beneficial when a compiler (such as GCC) supports multiple languages with a single internal data structure and unified back end.

This data structure includes a list of filters drawn from the project file. Each filter is loaded in turn and the data chain is made available to it. The filter is a process that can pass the data through or choose to modify it. The filter may also read and write environment variables to

communicate with other filters without needing to imply their existence, domain or operation. Having identified a system property a filter can use the Shards system to store it so that other filters who know the meaning of the data can retrieve it. Thus filters do not need to communicate directly but can instead use the data chain or the Shards system as a medium.

Finally a filter may also add additional filters that will be run. A filter is largely custom written and designed to encapsulate an atomic unit of functionality within one domain of system construction. The filters will gain from access to an interface library allowing it to communicate with the Shards system infrastructure. It will also have a library of frequently used operations and a general filter template to make construction easier.

It is believed that the operation of a filter will have a regularity in its internal operations. These will be initially captured by libraries of utility functions. The discovery of frequently accessed operations allows the possibility of a filter being described using a higher semantic level notation and then automatically converted into an executable form. This is directly equivalent to Lex having a high level notation to describe parsing operations which can then be automatically converted into an executable form. The general process to enable this was discussed in section 5.5 and referred to as meta-filters. From a purely mechanical point of view they do not offer new capabilities for expressing operations. The advantages of the concise expression of operations commonly used across multiple system development efforts also better suit a much later iteration of the Shards system.

Some filters exist only to provide a single name by which an integrated family of filters can be scheduled in the correct order. This will occur when a filter uses or reuses other filters to provide some or all of its functionality. This process may also be nested where some of the filters being used are themselves constructed using filter reuse. For both the operating system designer and the filter implementer it is more convenient and concise to have this detail concealed behind a single filter that identifies the functionality being offered. In the purest form this means the top level filter (referred to as a *filter family*) focuses on encapsulating the subcomponents and does little more than add them, in the correct order, for execution, which is a process these filters may then repeat when they are called. This reuse is highly desirable in order to make filter construction more efficient, allow filters to be constructed in a modular fashion and reduce the total number of filters within the Shards system. At the same time this encapsulation does not limit the operating system designer who could manage the sequence themselves or use nested

filters directly if they wanted more control.

It is expected that most filters will simply pass the majority of data through without any change. Only code that will influence or be influenced by the operating systems operation and falls within the domain of the mechanism encoded by the filter will trigger an action. Filters may recognise a pattern of interest but not make changes directly, instead scheduling other filters or setting environment values to trigger downstream actions. Filters that do implement changes to the application code may involve the code being modified, the code being converted into a system call if the functionality is going to be provided by the operating system or the code converted into a binary object if complete control of the application's operation is required. The end result is that the modified application code and operating system code and libraries selected and optimised to support the application load will be delivered to the compiler for conversion into an executable system.

7.2 Capturing Information

The primary input to the Shards process is a project file. The project file is a list of filters, with optional arguments, in the sequence they should be processed. The arguments can be used to indicate the location of any other needed resources such as application code to the filter that will process them. This processing can involve the use of external tools or processes such as compilers, compiler front ends or tools like Lex and YACC as components. The processing will result in additional data being introduced into the shards system for further internal processing by later filters or as analysis results that filters can react to. The project file may also contain a number of environment variables which specify system aspects that filters can respond to directly.

The amount of data that can be derived from analysis of source code is an open question. On the one hand, it offers the best source of information for filters to derive information on the operation of the client applications it is being tasked to support in an optimal fashion. On the other hand, deriving intent from source code and making changes that preserve all existing behaviours is a challenging task. The supporting evidence that the idea is worth pursuing is the fact that optimizing compilers do perform this function and there is a great deal of research on the topic. The Shards system simply provides a framework in which such analysis may take place and be applied rather than offering a solution to simplify the process.

The principle of fall-through is even more important in this context. It is assumed that a small subsection of all the application code will be analysed. Studies on application influences on memory use (such as [LaRowe et al., 1991; Raina, 1992]) find that the information gained will be exact only for fine details and statistical for broad behaviour. Analysis for the purpose of optimisation does not have to be complete or perfect in order to be useful. The better the analysis the more definite the solutions can be but even partial analysis and some general statistics of usage patterns still have value. It also has a value the operating system designer can not easily provide directly, and certainly not in a way that can be reapplied to a changed application load or future system. The ability of Shards to directly use the results of source code analysis as input to the optimisation process could also motivate further development of analysis tools and matching optimisation possibilities. The current approach has many analysis tools generating a report which must be analysed carefully and implemented manually which reduced the convenience of their application.

A hybrid solution to application code analysis is the *hint* process where the application programmer provides optimisation clues intended for the Shards system in the application source code, for example as a structured comment that the filter recognises. This may enable automated analysis to be focused and improved based on this information. The disadvantage is it requires a close link between the filter and the source code. The advantage is that it is potentially much more exact for less effort than fully automated analysis can hope to be. Effectively this approach gives another option if the optimisation of the issue in question is very important but analysis code to automate detection is impractical or immature.

This approach also works where the application programmer desires to add new primitive features to the implementation language. This can not be expressed in the existing language but they can be used freely as long as there is a filter to recognise and handle them prior to final compilation. This will generally be handled in two stages if using a pre-existing compiler front-end. An initial filter that converts the primitive into a form that will be accepted by the compiler being used and produces an output that can be recognised by a later filter for more detailed handling if required. If the filters to implement this language extension do not exist then the compiler will identify it as a syntax error.

Some filters will analyse the usage of operating system resources found within the selected applications and apply this information in selecting and tuning the solution without making any

changes to the application code itself. Other filters may choose to modify the application code as either a full or partial solution to a design goal. A filter may even choose to fully convert the code to assembler when it needs complete control of the compilation process in the case that it has specific knowledge about the target architecture that must be integrated as part of a solution. It is likely that most filters will not require this level of control and will use higher level languages or link to pre-built library objects.

In practice one of the last filters will create temporary files containing any application code that has been modified. This code will then be presented to an architecture specific compiler for conversion to an object format to be used in building the application executable. These object files can then be integrated so that they will be used as part of the application build process.

It is worth noting that the Shards processing is not concerned with syntactic errors in the application code. These will be caught as normal when the code is presented to the compiler As such there is no benefit in duplicating this analysis within the context of the Shards processing. This also applies to code optimisations performed by the compiler back-end. In general these processes are sufficiently mature that Shards will not attempt to replicate them. They are also sufficiently localised that they are unlikely to compete with a Shards optimisation process. If conflict does occur either the optimisation would be disabled or a custom compiler backend would be required.

7.2.1 Internal Data Structure

The internal data structure for the Shards system must be suitable for automated manipulation but must also be information rich. There is no universal design allowing a parser or filter to know what information will be required by other filters in the processing order. It is for this reason that a great deal of information a normal compiler would discard in the interests of efficient memory usage, such as line numbers after syntax checking has been completed, are retained for the duration of the Shards' run. This also allows more informative feedback to the developer when processing does not proceed as expected, as such data items are ideally descriptive and self contained.

All universal systems like Shards run into the issue of defining an ontology. A filter may describe a data item for its own use but it cannot know that it is unique or that other filters who can access it via the data chain will understand its meaning and respect its expected

behaviour. There is not really an automated way to enforce this as filters are human constructs. The creation of a global ontology for all terms meaningful in the systems domain is a large and probably intractable problem, especially since its solution does not actually have a direct functional benefit. Instead it is expected that just as filters within a component will become the foundation for later experimentation and evolution they will also contribute to the terms and their meaning for the community who works with that component. The Shards system assists by making terms human readable and having a composite name-space so that each component can have its own name-space.

One of the techniques used to achieve this is the notion of the *atom* which has been inherited from an earlier implementation of Shards in the Erlang programming language [Armstrong et al., 1996] but which is shared with many functional languages. In the Shards usage, an atom is effectively a string used as an identifier of a semantic object. For example the atom "internal" is used as an identifier for data items that relate purely to the internal operation of the Shards system. The advantage is that the name gives human meaning as to the function without requiring this meaning to be expressed within the implementation code or even known in advance. The implementation allows efficient string storage and mapping of an integer to a string and vice versa. This means that any use of the string "internal" in an atom context can be guaranteed to give the same unique index number for efficient storage in data structures while only needing to store a single subset (a string difference) of the original string.

The Shards system gathers data into tuples which can be arbitrarily long and can have a complex internal structure. The first element is always a 3 element tuple with a consistent structure allowing the identity and thus structure of the data to be identified. This identifier is called the *head*. The first element of the head is an atom indicating the *family* of filters which which it is associated and which document the structure definitions of data items they recognise. The second element indicates which structure is being used while the third is a specific identifier, such as a atom or index number, of the piece of data. If the sequence of data items is sufficient identification (for example parsed data tokens will naturally be in order) then the identifier may instead be used to store data. The data elements may be dynamically deconstructed so that a filter can recover the type and value of the primitive data types used. This may be useful in representing raw or unstructured data. In the general case the head provides the information to identify the expected internal structure.

The family to which a data tuple belongs is not exclusive so a filter that understands the operation of another filter family may choose to create or manipulate data carrying a family atom other than its own. In some cases, such as the early parsers, this is the expected operation with the data item being created for consumption and manipulation by a potentially broad range of later filters. Filters are free to change this family line if they make sufficient changes to the structure and content of the data that it is better considered as a new type.

A family has a specific meaning in the implementation which also serves to protect against name collision. Specifically a filter family name is expected to be the same as a filter distributed with the Shards implementation or a new project specific creation. The existence of the filter effectively reserves the name as author added filters would have to exist within the same file-space. The family filter may serve as the root filter for a family of subsidiary and associated filters all of whom will use the same family name and manipulate the same data items. The family filter is recognised in the current implementation through not having any underscore characters within the name. Thus "parser" would be a family filter name while "parser_c" and "parser_internal" would be filters that are members of the parser family. The family filter also has some additional implementation requirements. It takes responsibility for queuing its sub-filters so that adding the filter "parser" would involve it scheduling some or all of its subsidiary filters based on the needs of the input. For example a parser filter might look at the source language of the input and load one or more language specific parsers. This allows the user to treat the family as a unified object and allows the parser implementation to be modified without breaking the user's project files. A family filter will also provide some externally accessed functionality that helps present the family as a unified object. For example it will convert any data items owned by the family into printable strings for output and debugging purposes.

A large portion of the potential value of the Shards system would be consolidated in established and proven filter families that would be included with the software. These would provide a standardised toolkit, with known names, that could be used as a foundation by the prospective developer. The actual construction of a fully featured collection is outside of the scope of this thesis and would require a community of contributing developers. This would also require some organisational control on when filter families become standard and thus can be said to have a claim on their family name. It may even be desirable to use a naming scheme, such as prepending a leading underscore (e.g. *_name*), to identify standard libraries so that future libraries are

less likely to have naming conflicts with a user's privately constructed filter sets.

The Shards system is permissive and does not restrict the ability of filters to freely modify data items. This includes modifying data items associated with other families as well as changing the internal structure of the data. It is expected that filters may extend or interact with a family and thus need to both read and manipulate data items belonging to that family. They may also choose to reuse some of the member filters without engaging the full sequence. This is desirable behaviour as it represents effective reuse of existing components. The normal software engineering recognition that coupling components in this fashion can make the larger structure more complex and fragile is still relevant and should be respected by the author of a new filter. The filter author must not change the structure of the head as all filters rely on this having a consistent structure. Like a compiler malicious or defective filters are faults that must be discovered and corrected.

The Shards systems reserves the family name *internal* for its own use in identifying Shards system management objects on the data chain. Thus there will not be a filter family named *internal* or using it as a family name. These elements will always be flagged as being system elements and thus will not interact with search operations unless specifically requested. Filters are capable of interacting with internal data if they need to but since the data is primarily administrative there should be little reason to do so.

The operation of the Shards system uses data items that are visible to filters and could be read or modified. These include processed filters, a data item indicating the current point of processing, queued filters and the division between filters and data being used in processing such as parsed source code. Environment variables are held in memory by the shards system as they are global in scope. Filters will also use internal variables and data structures during their operation which will not be represented on the data chain.

7.2.2 Additional Structures

Within a data item there are some additional items used to store structural information about the interaction between data items. These are generally referred to as *flags* since in the interests of space efficiency they will be implemented as bit flags. These flags are related to grouping multiple items together (group and insert flags), marking some data items as not connected with the initial data (system and context flags) and about the larger structure in which the data

items are held (link, fold and external flags). This section will present each of these flags in more detail. Flags are set through Shards API calls and are not manipulated directly as parts of the data structure.

A data group is a collection of items that are generally added at a similar point in time and used together as part of a sequence of filter operations. While the group will remain in the data chain its relevance is expected to be reasonably short lived. The long term impact is more likely to be through creating a data chain entry that documents the results of the operation. This is why the group mechanism does not support sub-groups or protection against being disturbed by later operations on the data chain. These facilities are available through use of the *fold* operation which will be described later.

The group and insert flags are used together to combine multiple data items into a single structure. The specific cases in which this is required are largely under the control of the filters doing the transformation. There is an expectation that most filters which generate multiple data items as a result of their processing will mark them as a data group for clarity even if they will not manipulate them again and do not expect a downstream filter to do so. For example a parser that is going to convert a single source line into multiple data items should indicate their original connection through using the group mechanism. This does not change the structure of the output but allows for the possibility of later filters that want to operate on all the output involved but would find manually discerning the group members difficult. In implementation terms a group is defined as a number of sequential items where the group flag has the same value. Thus a change in the flag value indicates the boundary between two groups. The Shards system automatically manages the application of the group flag. It could be manipulated directly with *dc_setFlag()* but there are no current conditions under which doing so makes sense. Instead when new data is added to the data chain using *dc_emitData()* the Shards system will make sure the group and insert flags indicate the data was added as a group.

The insert flag is used when a new item is being added to the chain either as a new addition or through partial transformation from an existing group. In this case the new content retains its current group value but the insert flag is set to indicate it is actually a separate group. The Shards system manages the precise values used as it is dependent on the state of the groups bordering the new addition. When adding a new group in place using the insert flag it also checks the bordering groups to see if they also have the insert flag set. If the new insert shares

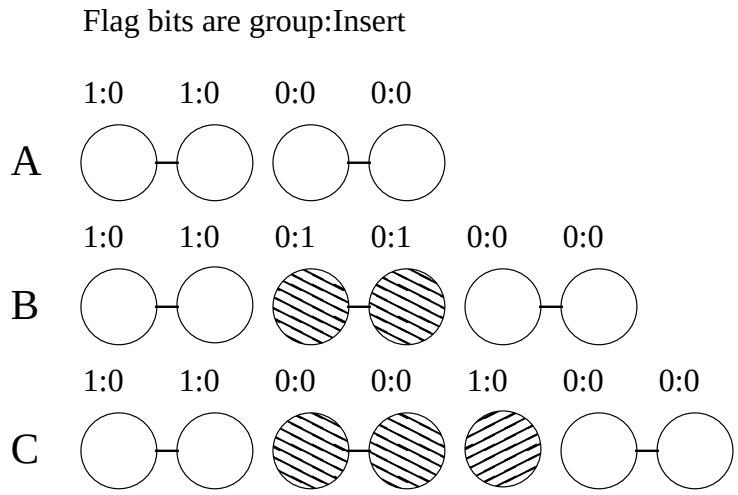

Figure 7.1: Group and Insert flag example

a border with an existing insert group then the new and the existing group will both drop the insert flag and make sure their group bit values are opposed to indicate the group boundary. If both borders have the insert flag and the same group flag value, for example if the new addition is in the middle of an existing insert, then the new content will be given an opposed group flag value so that it may act as a divider. This allows all bordering groups to reset their insert flags. The situation in which the new content is added between two groups with different group flag values and the insert bit set is not possible as they would have dropped their insert flag when the second insertion occurred.

In Figure 7.1 a graphic representation of groups within the data chain has been provided. On line A there are two groups with the boundary delineated through a change in the value of the group bit. In line B a new group is added in the centre. Since it cannot use the group bit to indicate the border with its neighbours (since they are using both group bit values) it sets the insert bit to allow the three different groups to be discerned. The group bit is also set to differ from one of its neighbours. It does not really matter which neighbour is used to set its group flag opposite to (left or right) as long as it is consistently implemented. In this case it sets the group flag to the opposite of its leftmost neighbour. The addition of a third item in line C uses the same logic and sets its group bit to 1. At the same time the implementation notices that one of its adjacent partners has the insert bit set and knows the insert has resolved the sequence (both borders correctly marked) and thus clears the insert bit. If the new addition was in the middle of an old insert block it would know this by there being insert bits set for both

its neighbours. In this case it will remove the insert bit from only one of the two partners using the same direction as the group flag is set relative to.

The system and context flags are used to indicate data items that are not generated from the input data and will not directly cause output to be generated. The primary use is that it allows filters searching for patterns in the input to automatically ignore these derived data items as "noise" that would confuse the goal of identifying structures contained in the original input source or added as output by another filter. Thus while the Shards system provides various search functionality over the data chain these searches will ignore items marked with the system or context flag unless explicitly told to include them. In a similar fashion searches can be organised so that they only look for matches with items bearing the system or context flags and ignore all other data. If the search pattern contains data items marked as being context or system items then this quality will also be checked for a viable match without making all system or context items visible. These flags are primarily about "noise" management. A pattern derived from the source may have a great deal of derived information separating its component parts which can make searches and manipulation complex. These flags allow primary and derived input to be separated in order to enhance the process of searching over the data chain.

The more specific meaning is that the system flag indicates a data object used by the Shards system as part of its internal management. In general it is expected that most filters will not find many reasons to examine or manipulate this data and it can be safely ignored. Context data is information derived from or supporting the primary input. For example the results of analysis on a data structure may be written into the data chain for the use of later filters that will do further analysis or act on the results found. This is an example of information that may be considered context information. Another example of context data is white space and file names from the original sources. This is generally not meaningful from the point of view of a programming language and thus most searches will ignore it. However if a structure is white space sensitive or when the Shards system wants to return source file and line number information for debugging purposes this information is important and thus is retained within the data chain.

When new data items are added to the data chain the group flag is used to indicate their separation from the surrounding data items. If the new items being added are identified as being system or context data then the group flag works somewhat differently. A context data item will

share the group flag of the data items it is being added as part of. If all the data items being added are context items then they share the group flag of the data currently being processed rather than forming a new group. This is both so they are linked to the data being read and so they do not break existing groups. System items are considered to be group independent and are neither members nor interruptions to existing groups. This behaviour can be over-ridden if the data item is flagged as being both system and context in order to indicate a system item whose meaning is dependent on the data items being added or processed. In such a case the data items group flag will be set and acknowledged. This means a transform that consumes an entire group will also cause this data to be modified or discarded as the data to which it refers no longer exists and it is dependent on that data for its meaning.

The group membership of the context or system items becomes relevant when the data with which they are grouped is transformed or removed. In general system flagged data items will be retained (or removed directly by the Shards system) while context flagged data items will be dropped. As an example if three lines of input source, which would have context data items giving the source file line number, are converted into a single new data structure then the actual line numbers are lost. This is acceptable behaviour since the new data item is not specific to any of the original lines and source line data outside of the transform can still be used to construct a source line range which the new data item represents. In most other cases it encourages context information to be generated and acted on either before or as part of the transformation it is informing. Once again this is desirable as it limits the accumulation of possibly stale and irrelevant context data. It is expected that all context information will be discarded as part of the final output step. A system may choose to run a late order filter that looks for context items which should have resulted in transformations (and thus their removal) and identifies their continued existence as evidence of an error in handling.

The final set of flags are external, fold and link and all of these relate to the structure of the data chain itself. The external flag is the simplest and is used to identify two qualities about the constructed data item. The first is that the data item occupies a substantial size and the second that the data item internals are not informative or likely to be manipulated directly. If an operation is performed it will only be to change the references to the data item or its position in the data chain. As such there is little value in storing all of the data in the data chain. This flag can either be applied when creating a new data item, added to an existing data item or

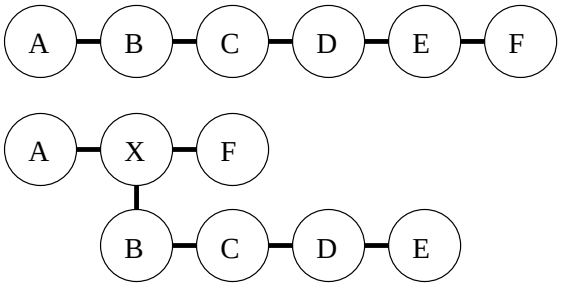

Figure 7.2: Folding the Data Chain

automatically added by the Shards system if required. The practical application of this flag is that it indicates the data item may be moved out of the in memory data chain and written to external storage if memory usage must be conserved. This will leave the head with the external flag set and any information needed to recover the body of the data item on access. The Shards system will take responsibility for managing this resource. An example is when the data chain contains assets such as bitmaps or large binary objects. These can be sizeable and are unlikely to be analysed or manipulated other than as an entire resource unit. Any filters that take part in generating output must support the possibility of data items being external.

The fold flag is used to temporarily remove potentially large but uninteresting sections from the data chain. This allows easier analysis of the surrounding environment and means the section of data can be moved as a single unit if required. In theoretical terms the fold represents compressing one or more data items into a branch from the main data chain. This means that as many data items as are required will be represented by the single data item which connects them to the data chain. This fold data item will be ignored on searches. The actual implementation is achieved by creating a new data item with the system and fold flags set which acts as a root for the new branch. Alternatively the range of included data items could simply all be flagged with the fold bit and treated as a single logical entity that once again is not included in searches. Folds are not meant be long term constructs as they represent structuring the data chain as part of an analysis. A filter family that makes use of folds should run the system provided unfold() command on completion which will expand and remove all folds in the data chain. There is a variant of the unfold command which takes a family name as an argument and will only expand folds identified as having been created by that filter family.

In Figure 7.2 a simple graphic representation of a fold in the data chain is presented. The data items B to E have been selected and used in a fold command. This creates data item X

which exists only to serve as a root node for the old chain and is flagged so that it will not appear on searches unless specifically requested. This means that for most utilities operating on the data chain it now consists only of the items A and F but can be easily reconstituted to its original form when needed. This can be convenient for removing material judged uninteresting and obscuring determination of higher level patterns. For example A and F might be the start of a function block but the filter is only interested in the function signature itself. It becomes easier to analyse function call sequences if a filter (ideally reused) can make invisible those code elements not related to function calls without actually losing or having to manage the data.

7.2.3 The Link Flag and Nested Chains

The previous subsection introduced the concept of fold flags which indicate a simple branch. This branch is intended to be temporary and not represent a permanent modification of the data chain. It is also always a simple branch with a single linear data chain connected to the main data chain. It cannot be nested, and so there will not be a fold within a fold. This allows the fold mechanism two implementation possibilities, either physically rearranging data chain connections or flagging all participating items to have identical semantics. In either case the actual user provided data is not modified.

The link flag is used to indicate that the data item is involved in more fully featured branching. A data item bearing the link flag can contain one or more data chains within its body. These data chains may themselves contain further link flagged nodes up to an arbitrarily deep level of nesting. The structure formed in this way is considered permanent and meaningful. In other words it makes sense for the semantic object indicated to have data chains as subcomponents by nature of what it represents. The best known example of what could be constructed using this facility is a parse tree where the arguments to an operation could themselves be nested sequences of operations.

A filter that wishes to ignore the structure can use the get_link(*index*) call on any data item that has the link flag set. This call can be used to iterate through the components of the data structure that represent links to nested data chains. This call can be used even if the filter parsing the data chain is not aware of the internal structure of the data item. The Shards system will automate calling the family filter that does understand the structure and it must supply an implementation of this function. This allows filters to traverse any nested branches

within the data tree and thus treat the entire structure as if it was a linear construct.

The ability to treat a parse tree as if it were a linear construct seems paradoxical. In a regular compiler transforming the input data into a syntax tree occurs almost immediately and the resulting structure will be retained until the tree is converted for output or handling by a compiler back-end. The Shards system however is less focused on converting the code into language constructs as this will be handled in the final compilation by one of the existing language specific compilers. The Shards system also uses the data chain to gather data that is not specific to a single programming language such as pass through hints from the application author (see Section 5.7), results of previous filters and context data generated from analysis. Indeed the Shards system is more likely to focus on the type and frequency of operating system relevant operations being performed over the actual structure of the language in which they occur. Thus it is quite probable that many filter sequences may choose not to construct or use a parse tree (or use only partial segments) as required. Using a linear data chain allows filters to be more generic as they do not require the substantial amount of language specific knowledge required to manipulate a parse tree and can instead focus on the operations or system calls relevant to their domain.

7.3 Expressing Transforms as Filters

The filter is the primary mechanism used in the operation of the Shards system. Conceptually filters are not complex. A filter is able to read the data existing on the data chain and either extend or replace it. This section will discuss some of the details involved in the implementation of the Shards filters. The focus also includes making filters convenient for future implementers so that the Shards system can be extended to meet future needs.

The processing within a filter is performed using an existing programming language. An established programming language is expressive, proven and familiar. Experiments indicated there was little benefit to generating a new custom programming language. The chosen language interacts with the Shards system through a standardised API. This would allow the filter implementer to determine the most convenient language for their needs as long as it had a compiler for the target system and a wrapper to call the Shards API (which is currently implemented in the C programming language). This allows the filter to use all the expressive power and convenience combined in the language of choice. The weakness of the approach is that

simple filters, of which there are expected to be a sizeable number in any project, do not need this flexibility and may be less efficient to construct. This weakness will be addressed in the section on filter expansion.

The Shards systems uses the dynamic library loading functionality available on Unix systems to be able to extend the list of available filters at run-time. This also means that all filters do not need to be written in a single programming language. As long as the language is capable of writing dynamic libraries that can be accessed by the POSIX.1 specified functions they can be used in the Shards system.

The filters are source files within a directory known to the Shards processor. It can be either explicitly defined or provided as an environment variable. This directory has one sub-directory matching the filter's family name to reserve the namespace. If the source for the filter has been included (binary filters are possible) it will exist in this directory and may also occupy other directories for sub-filters. These sub-filters will have the prefix <family name>_ to identify them as being sub-filters, though other projects that understand their function may be able to use them independently. The source in this directory will generate one or more library files, which will exist in a directory named "bin". These library files will begin with an underscore if they are part of the standard distribution. This allows file names not starting with an underscore and not already reserved to be used for user-created filters. The first filter generated will match the name of the source directory and is the *master* filter. It knows how to queue its subcomponents (if required), contains templates for family specific data items and implements some functions to allow the wider Shards system to access these data items without having to understand their specific structure. This is currently used mostly for link decoding and generating readable representations of the data item for diagnostic logging. The master filter may contain all the functionality to read and transform the data chain in which case it will be the only filter within the family in addition to being the master. In practice any non-trivial filter is likely to use sub-filters to implement functional subcomponents. This makes the individual files simpler, organises the functionality within the filter and allows for reuse of sub-filters.

The filter named Shards_init is reserved by the Shards system. This filter is responsible for queuing any filters needed to initialise the Shards system and set up the data chain. The intention of this approach is that the Shards core will be small and change its functionality rarely. Evolution in how the Shards system initialises and what filters it uses will be contained

within the Shards_init filter. This allows a newer Shards engine to run a project dependent on an older behaviour by redirecting the start-up to one of the older versions of the Shards_init filter. The Shards_init filter is not project specific but will queue filters to parse a supplied project file which will contain initialisation data and the project filters that are required to be run.

The initial filter and the project file will store some global environment variables. These are stored within the data chain in the current implementation although using a database for storage is also viable. They are accessed using the function calls *dc_getenv()* and *dc_putenv()*. A data item is a self-descriptive multiple-type composite data structure constructed using the Shards API (see data.h). It is expected that all environment variables will have two strings (actually atoms) forming a name-value pair as their first two members. There may be additional data items if required. Filters using the data item can assume the existence of the name-value pair and based on that information determine if they expect to find additional data items.

These global environmental values are intended to be context information that any filter may choose to access. They represent information on the target of the project that the filters can use to correctly tailor themselves. This may include hardware information such as the processor, number of processors, available RAM and even cache structure. The list of possible entries will be externally standardised so that if a name such as "processor" exists the filters will have expectations on which values may exist and what those values represent. A name may be absent as not all possible values will be relevant to all projects. In this case the filter may either use a default value or make its own determination of the correct behaviour when the specific value is unstated. The current Shards system does not have a fully matured version of these variables as constructing a complete table of potentially meaningful data is a substantial task, not required at this early stage and better developed through evolution during use.

The filters can also store information intended for communication between subcomponent filters. Filters will generate their own internal data during processing but this will not persist once processing is complete on that filter. A filter family can use a number of mechanisms to store working results and communicate. The Shards system provides an environment wide data store using tuples similar to those in the data chain from which filters can read and write values. These can either be truly global or be identified by the tuple header as belonging to a specific filter family. Filter family specific information is stored using the calls dc_getenv(*family,data*) and dc_putenv(*family,data*) and is handled in a similar fashion to the global environment values. The

main difference is that filter data will not be returned from the global form of the get operation or from a get with a different family name. Other filters that fully understand the meaning of the data may access it by providing the appropriate family name even if that is not their own family name. The Shards system does not restrict or control access or manipulation of the data as cooperation and extension between filter families is expected. The responsibility lies with the filter creators to make sure their use of these data items is uniform and compatible with the intent and design of the module that created the items originally.

Filters can communicate by writing data items into the data chain. There are multiple functions to do this depending on the specifics of the operation. These items will be have a header giving the identity of the family that wrote it but as with global data other filters that understand the data can modify it. This allows variant and extension filters. Data placed into the data chain is harder to locate and easily edit and so is less suited for general calculation or communication. The main advantage is that it has an explicit position in the sequence of the data chain. For example this could relate to code being analysed or data that is shaping system construction. It is also expected that filter families have a clean up filter where data items added for communication and sharing state are removed. The Shards system provides functions to remove data items via pattern matching to assist in this process.

It is expected that filters will also contain and be connected with documentation. A large part of the value of filters is that they can be organised into cooperative sequences. For example many filters might reuse a set of filters that know how to parse application source code and convert it into a sequence of data on the data chain. As long as the items it can generate are documented they can be used as input for specialised analysis filters that transform or extend this data to include terms used by a filter family, which will themselves be documented. This allows the ability to add new filters into an established family which enables extensions and variants to be created. A filter should be a unit of code within the Shards system, a part of a filter family that expresses some functional element and potential material for reuse or extension. The filters are complex enough that determining this through examination can be slow. Good documentation will enhance reuse through being able to understand the intent, operation and data items recognised and emitted by the filter. It is expected that much of this documentation will be shared. For example creating and documenting a set of data items that represent all fundamental memory actions creates a resource that could be used to enable all filters in that

area to work together and simplify their individual documentation. Most filter families are designed to work closely together and are also likely to have a single document for data items common to all filters within the family.

It is also expected that filters will start at a relatively broad level and with incomplete coverage. The initial implementation of a Shards process may only examine the context of a small section of an operating system and make a small change to a the configuration options of a build or generate some source code files to be included in the compilation. It may also take the form of only a single filter with most of the complexity in the code within the filter rather than the interaction of filters. If there are no opportunities for optimisation, or no time to take advantage of them, there is no need to write additional Shards filters or wrap the entire operating system within the Shards system. This is intended to make the development of an initial Shards system a relatively manageable process.

The intent is that this structure will get reused in future versions of the operating system or other operating systems. This will encourage it to naturally evolve. An optimisation possibility may require more analysis of the context or application load, more calculation and more changes in the underlying system which require a more sophisticated filter. As the filter grows and internal functionality is reused there is reason to start breaking the initial filter into finer functional blocks each of which can be a separate filter. Where an optimisation possibility has several possible approaches these can also be separated out into filters so that strategies can effectively be swapped in and out of the system if needed. This also allows easy experimentation of how the different strategies interact with the evolving system and other filters in use. These strategies then have the possibility of being transplanted to other systems. As the level of interaction grows filters communicate and ideally cooperate to a greater extent. Allowing the Shards system to grow based on the needs of the project is important to making it more practical. Having to represent the entire system within the Shards system before it could be used would require a very large upfront investment in effort that does not directly push the construction of the system forward.

7.3.1 Filter Example

An example filter is shown in listing 7.1. This example does not do much and has no error checking in the interests of brevity. However it provides a general framework which most filters

will follow. This code uses some of the calls provided in the Shards headers which are reproduced, in enough depth to support this example, in the Appendix.

Listing 7.1: Example Filter

```
int sample_filter(dyndata* args)
{

    dyndata* data;
    int argc;
    int argv = 0;

    // set up dynamic data structures
    data = data_new();

    // see if we got any args, we may get one integer
    if (args!=NULL){
        argc = get_int(args);
        if (1 == argc){
            argv = get_int(args);
        }
    }

    // set up a search for an element
    findFamily(atom_put("example"));
    findType(atom_put("data"));

    // find all the occurences and add the data they hold
    // to the argument.
    while(DC_SUCCESS == dc_search()){
        data = dc_getBody();
        argv += get_int(data);
    }

    // search complete, so set it as an environment variable
    // this may be all an analysis filter would do.
    data_clear(data);
    data_put_int(data,argv);
    dc_putEnv(atom_put("example"),"total_value",data);

    // store it in the data_chain as well to demonstrate
    // so find where the total is stored. Can reuse search params
    dc_searchReset();

    while(DC_SUCCESS == dc_search()){
```

```
        data = dc_getBody();
        if (atom_put("data_total") == get_int(data)){
            // found the total. going to replace it so "consume" it
            dc_consumeData();

            // write new version
            data_clear(data);
            put_int(data,atom_put("data_total"));
            put_int(data,argv);
            dc_emitData(atom_put("example"),atom_put("data"),data);
            dc_next();

            // set up another filter to do more (no arguments)
            data_clear(data);
            put_charA(data,"another_filter");
            dc_filterAdd(data);

            // also set it as the current last filter to run
            dc_filterAppend(data);

            // only expect one data_total
            break;
        }
    }

    // all done!
    dc_filterEnd();
}
```

The filter users a lot of Shards API functions but it is hoped the general sequence is clear enough. It makes extensive use of the *dyndata* type which is a dynamically constructed data structure made up of a list of type and value pairs. In this example the Shards filter will have been loaded as a dynamic library and given an argument list taken from parsing the project file. In this case the argument is an integer. In the background the Shards system maintains a pointer into the data chain giving the first data item after the filter list. This is not given as an explicit argument to the Shards function calls as they all reference this data structure internally. The filter searches for tuples of <example, data, X> and adds the X value to the argument. This is then written to the environment space and into the data chain which is entirely redundant but demonstrates both approaches. In addition it schedules two filters to continue this analysis project. This models the expected behaviour in which early filters in a

sequence focus on analysis and marking out the location of interest. Later filters will use this information to inform and target any substantial change to the data chain. When the filter ends the Shards system will repeat the process with the next filter in the data chain, which we know will be the filter called "another filter".

The use of the atom_put function may also seem counter-intuitive and is another remnant from the original Erlang design. The Shards process makes extensive use of strings as identity names which is similar to the Erlang use of atoms. The large number of strings take up substantial space and will mostly be used to test for string equality which can be compute expensive when performed over a simple data structure. Making this basic operation a Shards API call allows an optimised data structure to be used for all occurrences of this common operation. This also means that strings in the data chain are index numbers rather than literal strings.

The output sequence may also seem verbose. This is partly because the Shards system can exhibit quite chaotic behaviour when filters fail to work properly and then other filters react to their errors as input. In a similar fashion it can be difficult to trace back to which of the filters caused the initial problem. As a result the Shards design calls for the logs to record the changes made to the data chain by each filter. The record of all changes associated with the execution of one filter is referred to a *delta*. In order to accurately capture the changes made various Shards system calls should be used to process and modify the data chain. So in this case dc_consumeData() states we are considering replacing the item and effectively removes it from the data chain. dc_emitData() writes new data items to the chain. However the change is confirmed only when dc_next() is called rather than dc_consumeData() and the change written both to the data chain and the logs as a single transaction. Alternatively if dc_rewind() is called then the change is aborted, consumed items returned to the chain and the read head set at the point the first consume call was made. This allows multiple comparisons to be made on the data that is being considered for replacement. Making this process explicit was partly because automatically gathering changes and splitting them into transactions proved to be reasonably complex in practice.

The main difference with a real filter is that it will involve much more complex sequences and computation. In this simplified example the search is for a single data item with a specific identity. Filters become much larger when searching for longer and more varied patterns within the data chain. This is generally done by building up a search string with multiple elements to

Listing 7.2: Filter Sequence (Pseudo Code)

```
find sequence(pattern_indicator);
if (pattern_indicator == pattern){
        decision = calculate_action(pattern,stored_state);
        switch(decision){
                case: schedule_filter(filter_name);
                case: access(stored_state);
                case: modify(stored_state);
                case: write(pattern);
                case: consume(data chain);
                case: write(data chain);
                case: custom_c_function();
                case: end filter;
        }
repeat (search);
next_filter();
```

find a starting point and then manually stepping through the data chain to confirm a pattern of interest. While the example has only a single action, a real filter will recognise multiple patterns and have multiple conditional responses. Once the pattern has been identified, analysis, along with drawing information from environment variables and previous patterns found, can quickly become complex. The implementation, once the data has been recovered, is performed in traditional source code whose operation is largely invisible to the Shards system unless it requires a Shards API call. If changes to the data chain are required then these will be performed using Shards calls so that they can be tracked.

In general the model of a filter is to find a pattern in the data-chain, determine whether it is a pattern it recognises and is relevant and derive information from the body of the data items making up the pattern. A filter will generate state data either internally, on the data chain or in a Shards environment variable. There may be multiple analysis filters which will build on the results of earlier analysis. Once analysis has determined an action, filters can modify the data chain or write to an environment variable with the result and determine which filters to schedule in order to carry out the change. The general pseudo code representation of a filter action sequence is shown in listing 7.2.

7.3.2 Meta-Filter Expansion

A filter created using a full programming language has a great deal of flexibility and expressive power. However this may be excessive if the actual operation being captured is relatively simple. It is expected that there will be a need for many simple filters that represent basic and frequently reused transformations. In such a case the extra capability of a full programming language may simply mean more work for the filter creator. It may also make it harder for someone looking for reuse opportunities to quickly grasp the operation of the filter. Simplifying the creation and understanding of filters would be convenient to users of the Shards system.

There are a variety of ways in which functionality can be packaged for use within the filter code. The methods that are commonly used in software creation remain valid. A rich selection of library calls can be provided either by Shards or by user created libraries. These allow complex and proven functionality to be conveniently accessed so that it does not need to be recreated. For example many filters might involve scanning the data chain searching for a pattern and then replacing it with another data item. This could be expressed as a helper function. These API operations can be ported to multiple programming languages so that the API itself can be considered language neutral or universal. This allows a programmer who does not know the language in which the filter is expressed to still derive clues to its function by observing its use of known API calls.

This process can be taken further. If a sequence of operations is extremely common it could be compressed further using a customised high level scripting language. As an example, the action of finding a pattern and replacing it could be expressed with a single line of a regular expression based scripting language like Sed or AWK [Dougherty and Robbins, 1997]. This would allow a very concise and efficient expression of a filter's operations. Development could be driven as implementation efforts reveal frequently used sequences that make a good target for conversion to a script style expression. This more concise notation would then be executed on an interpreter within the Shards system or expanded into code that could be compiled as part of the filter.

Conveniently, the Shards system itself is well suited for precisely this form of expansion. In this system the instructions for compiling the filter code into loadable libraries would include running Shards on the filter code itself. This Shards processing would search for script blocks and extend them into the implementation language being used. This expansion would also include

calls to the Shards APIs as needed. The files would then be compiled to produce usable filters. This allows the existence of multiple filter sets to support multiple scripting languages in order to avoid the design complexity of needing a single, complete and universal script language design[2]. These expansion filters can be extended to support new scripting operations as required. They can also be extended to allow translation to more implementation language targets allowing the script operations themselves to be considered language neutral.

Since a filter is implemented using a full featured programming language it can potentially do any sort of computation once it is loaded and executed by the Shards system. In practice however all filters have a specific goal and are likely to have some similarity in how they are structured, and will use the provided Shards APIs to achieve their goals. They will create artificial data objects described as *triggers* which represent a pattern they are seeking in the head of a data item contained in the data chain. Triggers will work much like a regular expression containing sets of atoms and operators for logical variants such as data elements not containing these atoms. They may also be incomplete to indicate that for some of the descriptors in the data element's head any value is acceptable.

For example, the search for a data element being used by the Shards system itself (the *internal* filter family) that is not a parser type but which may be any third sub-type and is flagged as being a system item could be expressed as {{internal,ˆparser,},system}. A sequence of these triggers would be deployed to find patterns of interest within the data chain. In the simplest implementation the filter author would have to manually code the comparison between this trigger and the current data item being read to see if it is of interest. In the Shards implementation the trigger is used as the input to a Shards API which takes care of iterating through the data chain and performing the comparison precisely because this operation will be extremely common. A continuation would involve being able to write the trigger exactly as it appears in the text above as part of a script and having a Shards filter convert it into code and function calls.

Once a match has been found, the Shards system will call a user supplied function that will more closely examine the data found and determine the operations needed. The end result will be either nothing being done or modified data (possibly containing parts of the original data

[2]The development of a universal, powerful and convenient language for scripting Shards filters would be a very desirable goal and an interesting research project. However it is not required for this initial implementation. In addition the imposing complexity of trying to design such a language is multiplied when the Shards system is itself under development.

found) being written back into the data chain. If the operation is reasonably simple calling a user supplied function is a relatively heavy overhead. For example if we simply wanted to user the trigger defined above and rename the subtype it would be extremely convenient and eminently implementable to expect the sequence below (where the unstated elements indicate no change in the data) to be capable of automatic expansion:

{{internal,^parser,},system} -> {{,,*new_name*},system}

It is expected there will be a point at which the challenge of supporting complex operations starts to conflict with the simplicity and convenience of the scripting language. That point could be at quite a high sophistication level and could use logic programming techniques to express complicated ideas in extremely concise forms. Exploring the boundaries of this domain is not the focus of this thesis. The advantages of a powerful filter expansion facility would be convenience when a large number of filters have been and are being written, which is more likely to occur in a production system.

7.4 Automated Application of Transforms

The general sequence of a Shards project starts with the shards_init filter. This loads a number of project general filters and global variables. At this point the data chain can be divided into three separate areas. The Shards system automatically manages the boundaries between these regions. It also maintains an indicator of the last item on the chain that was read for those regions that have a linear progression. The actual implementation details are not relevant to the filters or filter implementers as they will be accessed through general Shards API calls.

One region is the environment variables section which is not linear. It may be ordered in some fashion to optimise performance but access will be through a look-up based on the name of the environmental variable. Names are unique so adding a value to an existing name will overwrite the existing value. Checking the return of a get operation can be used to check if a given name exists. The command *dc_clearEnv()* can be used to remove a name and its associated value when it is no longer needed.

The filters queue can be considered another region of the data chain. This is a list of filters and their arguments arranged in the order in which they will be processed. The Shards system maintains a read pointer so that the system can recognise which filter it is currently processing and which filter will be executed next. When the Shards system runs out of filters to execute the

Shards process is considered to have successfully completed. It is expected that the Shards_init filter will add a small number of filters primarily concerned with parsing the project file. The project file will add a number of high level filters indicating the components to be used in constructing the operating system. These high level filters, which are generally the family filter itself, will queue a much larger number of subcomponent filters when executing. It is these subcomponent filters that will do the majority of the work.

Filters are not expected to manipulate the filter list directly. It is not given to them as input and they will not be able to access that section of the data chain. Instead a filter should use the Shards calls *dc_filterAdd()* to queue a filter for subsequent execution. The filter is placed next in the execution list and will be followed by the list as it was prior to calling add_filter. If multiple filters are being added then the filter is responsible for ordering the calls to add_filter so that the progression is correct. Filters may also use the call *add_filterAppend()* which queues the filter at the current tail of the filter list. This is useful for such things as cleanup or post operation analysis functions which will operate on data built up over the entirety of the execution. It is expected that in practice many filter families will begin with some number of filters performing analysis functions, a smaller number making changes based on this analysis and then one or more filters that check for consistency and perform cleanup. A filter may also choose to re-queue another iteration of itself if its processing requires two passes over the data chain though in general this is an indication the filter may be better constructed as two sub-filters.

By default, when a filter completes successfully, the next filter in order will be executed. Completion occurs when the filter has iterated through the entire data chain. For efficiency and convenience the calls *dc_filterEnd()* and *dc_filterRedo()* are also available. The first indicates that the filter has completed its task and iterating through the rest of the data chain would simply be wasted processing. The second indicates the filter wants to run again which is in practice implemented by immediately setting the next item position to be the first item on the data chain. This effectively resets the processing of the data chain to its initial position but retains any changes that have been made.

Filters also have access to a small number of commands which directly affect the progression through the filter queue. The commands are *dc_filterSkip()* and *dc_filterReverse()* which do not modify the filter queue but effectively move the "read" pointer. Both of these commands take the name of a filter as their single argument. Filter progression will be modified so that the filter

with this name will be considered the next filter to be executed and progression will continue from this point. The difference is that *dc_filterSkip()* moves forward while *dc_filterReverse()* moves backwards through the filter queue. If the given name is not found then this is regarded as a fatal error and the Shards process is terminated. This allows the filter queue to implement cycles and conditional progression based on the results found in a filter. In theory, these are not required as reverse movement can be simulated by simply calling *dc_addFilter()* with the filters that must be repeated. The filter skip operation is equivalent to not calling *dc_filterAdd()* until the correct filters to use are known and then only adding those filters to the chain. These commands have been added as a convenience for those constructing filter families and to keep the filter chain shorter in the case of *dc_filterReverse()*.

A filter may also call *dc_filterAbort()* which indicates that a fatal error has occurred and further progression would be meaningless. In theoretical terms this is equivalent to moving the read head to the end of the filter queue, but in practice the read head is left as it is and progression through the queue ceases. This allows the user to know at which point in the filter queue processing failed. Some filters may choose to queue an abort filter and *filter_skip()* to it prior to aborting. This filter could for example do some additional analysis as to the nature of the fault or some cleanup work. The process log and a dump of the data chain including the environmental data and executed filter list will also be generated to assist in repairing the fault.

The third part of the data chain is the data the filters will operate on. This is initially populated by parsing the source code that represents the application load. In practice the input could be any material that the project wishes to use and has filters to parse and process. It could even be empty if the project is generating a system that is not optimising itself against an application load.

The expectation is that most filters will use the provided functions to access each data item in order. This allows freedom in the actual underlying implementation mechanics. There are also search functions so that a large number of data items can be automatically skipped until a potentially interesting data item matches the search criteria. Changing the data involves calling *dc_consumeData()* which effectively marks the data item as being considered for change and advances the read head. In this way a sequence of data items can be connected as being a pattern that will be changed.

The process of consuming data can end in a number of ways. The command *dc_cancelConsume()*

cancels the operation and means the preceding *dc_consumeData()* operations are identical to forward iteration. The filter can use *dc_emitData()* to write one or more data items which will replace the consumed data items. The emitted data items may contain sections of the original material but is still marked as being a replacement (and thus by default a new group). The first call to the iterator *dc_next()* is used to mark the end of the change. For example calling *dc_consumeData()* ten times, *dc_emitData()* twice and *dc_next()* will remove ten data items from the data chain, replace them with two items and progress the read head to the item directly after the change. The final change resulting from all the operations is also written to the logfile and can be useful for examining what a filter is actually doing which can be useful for debugging. The complexity of these commands is designed to capture what is removed and added to the data chain as a transaction, to ease analysis and reduce the size of the process log. The log is still likely to be large and can be disabled for a stable production system.

7.5 System Generation

There will be a number of filters concerned with generating and shaping the output of the Shards process. It might be expected that these filters would run last in the filter order and this is generally the case. In practice however any filter that modifies the data chain (outside of creating or manipulating system or context items which will ultimately be discarded) has the potential to be an output filter. The determination is dependent on whether or not other filters will modify, over-write, discard or revert the changes made. A filter that runs very early in the sequence may thus be considered an output filter. The actual process of writing output ready sections of the data chain to file can itself be a reusable filter.

The primary input to the Shards process is the project file. This contains the filter families whose selection express a design strategy for the system being constructed. Each filter family represents a concept to be integrated which could be aimed at gathering material, performing analysis or generating a system component that can be used in construction of the target system. Some filter families will incorporate all of these stages in the sequence of subfilters they manage. The more information a filter can automatically gather and use in making optimal decisions the more value it can provide to a system designer. This includes giving a high priority to information provided explicitly such as arguments given in the project file or environment variables. The range of possible sources of information is limited only by what can be processed. The existence

of other filters, the data generated by other filters or analysis of some facet of the target system are all valid sources of information. This information is used to determine the ideal output. This can also cover a wide scope and could involve analysis information for use by other filters, modification or creation of environment variables, modification of system assets or arguments to the build process. The most significant output is the selection of mechanisms (likely to be contained as internally stored blocks of code), configuration of parameters and generation of source code modules to be used in system construction.

An optional input to the Shards process is a collection of application code. This material can be used to gather information but the Shards process could also modify the code in order to generate applications customised to the needs of the system design. It is possible for the input and output to be identical. In practice it is expected that the vast majority of the code will pass through the Shards system unchanged. The majority of application code will be running within the process image constructed by the operating system but will not require interaction outside of that environment and thus is operating system neutral. It could be said that the traditional compiler which directly follows the Shards processing is responsible for optimising code execution within the process environment.

It can be expected that traditional and well tested system implementation methods will continue to be used. At the heart of the operating system will be the smallest possible number of core primitive operations. Some of these will run in a constantly present software core with high privileges. A larger number will run outside of this core either as loadable modules, services (the microkernel model) or as a larger monolithic structure where the operational core and the services are aggregated into a single executable. All of these services will be accessed by the application code through a variety of library interfaces. The possibility also exists that some operating system functionality could be integrated directly into the application to reduce the overhead of calling the operating system. There are many possibilities each of which comes with advantages and disadvantages that must be weighed against the strategic goal of the system being constructed. The Shards system focuses on making sure that all possible approaches can be expressed and automated in a single framework.

Practical application of the Shards system is likely to focus more on selection and tailoring than dynamic creation of operating system components used in the solution. Generating a new approach or algorithm that can be used in operating systems will continue to require the

experience and ingenuity of one or more highly skilled developers. If the new capability is truly optimal in all cases then there is no need for the Shards system or further development. In practice many mechanisms incorporate design trade-offs that give them a set of strengths and weaknesses which will interact with the environment in which it is applied. The Shards system allows this environment to be known in advance so that solutions which are locally optimal to this environment and design goal can be safely selected regardless of their performance in other environments.

The Shards system increases the flexibility with which the designer may target their solution. It allows the creation of highly customised solutions that have an advantage in a specific environment. A solution would only have to prove that it has a local or specific advantage rather than being a globally optimal solution. Analysis filters are then constructed to determine whether the given application load favours one solution over another. This could be followed by manipulation of the application source to integrate it with the chosen solution. The selection filter then integrates the selected solution into the build process for the operating system core. The Shards filter plays the role of a system architect making the decision of which solution provides the best fit, but doing so in a way that can be recalculated when the application load changes.

Even when selection between multiple solution possibilities is not required, the Shards system can still assist with tailoring. The designer of an operating system capability may find it easier to create an optimum solution if some design decisions can be deferred until the environment is known. This can be expressed using the Shards system by creating filters which examine the source code to perform analysis which can be used to tailor the solution to the specific environment encountered. This type of Shards filter allows the component designer to express context sensitive design decisions in a way that can be recalculated when the application load or system priorities (as expressed by the system designer in the project file) changes.

The Shards system also allows for the possibility of a null mapping. If an operating system subcomponent is a truly globally optimal solution (or the only one currently supplied) then the Shards system does not need to perform analysis. The usage contained within the application load will always result in the same solution so it becomes an automatic part of the system construction. The inverse occurs when an operating system capability exists only to support a specific usage. Shards analysis filters can be used to confirm whether that usage exists, and

if it does not, the capability need not be included in the generated system. This is interesting because there are a great number of operating system capabilities considered fundamental such as multiprocessing and memory management which may not actually occur in all environments. For example, a remote sensor may be constructed as a single process in which case operating system multiprocessing and memory management support are not required and may be exacting a performance penalty based on the globally reasonable assumption that all environments will require this functionality.

The final construction and integration of the selected components into an operating system, and the application code into executable form, is flexible. This could be performed through a set of Shards filters calling external tools such as compilers and linkers to create and package the required binary objects. Alternatively, and more practically, the Shards filters can be used to generate a build script to determine selection and tailoring. This has the advantage of allowing the construction of the operating system to be decoupled from the Shards system (through a default build script) if required, for testing or for parallel development. In cases where interaction through the medium of a build script is not sufficient, a Shards filter could directly output source code for an operating system subcomponent which would then be compiled by the build script. Even in this case the Shards system is more likely to output a properties or definitions' file to complement or complete pre-existing source code than actually contain or generate the full subcomponent.

The Shards system will also generate a detailed log file of summarised changes. This file contains any log messages generated during the operation of the Shards system. This will include when a new filter is scheduled. It also contains a record of the changes that had been made to the data chain when the filter completed. This is written automatically by the Shards system as it manages the data chain, filter execution and filters requesting read and write operations to the data chain using its API calls. This allows it to automatically calculate the differences. Since the data chain starts empty it is possible to use this information, with the support of external tools to manage the volume of data, to step through the effects of the filter sequence on the data chain. This is most helpful when a filter is being developed or when the Shards process has aborted due to an error in one of the filters. In the case of an abort the Shards system will also write a dump of the current state of the data chain at the time the abort was called.

7.6 Conclusion

The Shards system is a framework that aims to combine three elements in order to allow the generation of customised operating systems for environments that have specific and demanding needs. The first element is the designer's expression of the architecture, which is contained in the project file. This list of environmental parameters and filters is a concise expression of how the system will be constructed. This is the information that would be used if the designer wanted to share or compare his design with others. The second element is information gathered by the operation of the filters through either their own analysis or using external tools. The final element is the generation of a system by the filters automated by the Shards process. Since the system is auto-generated the connections between the input and the system generated may be difficult to analyse but this is acceptable as long as the process is reliable and reproducible.

The application load is a potentially large volume of software for which the system is being designed and against which it will be optimised. As such it is both a source of information and optimisation possibilities. The end result is a selection of applications and an operating system which is optimally adapted to support them in line with the design goals of the system.

Chapter 8

Example Environment

> In testing a general purpose object (be it a piece of hardware, a program, a machine, or a system), one cannot subject it to all possible cases: for a computer this would imply that one feeds it with all possible programs!
>
> Edsger W. Dijkstra [Dijkstra, 1968]

Software is a virtual construct created in response to human requirements. It is not bound by physical laws and the variety of forms and the structures in which it can be shaped is nearly limitless. As a result *proving* a substantial body of software to have an innate quality such as correctness or to be optimal is extremely challenging. This challenge is increased when considering a system such as Shards which exists as a process for constructing software. For this reason, the evidence will have to take a more applied form. However, constructing a complete operating system is a substantial amount of work that could easily consume many person-years of work. Implementation detail and large scope can also obscure the role of the Shards process. It is better to provide an example that explores a potential system optimisation opportunity and how the Shards system could allow it to be captured and integrated in the system development process.

The challenge is to pick a concept that commonly occurs in operating systems and focus on it. The concept should be of a scale where it would represent a single filter family in an operating system model. This is the scale at which the Shards system operates in constructing an operating system from components. It should also be a filter family that has multiple possible solutions with the optimal solution being determined by the system context in order to make the

experiment more relevant. This will generally be true of any operating system subsystem that involves application driven resource contention. This actually includes a number of subsystems as resource management is one of the operating system's prime tasks. Ideally the concept should also be relatively concise and self-contained to restrain the amount of supporting context information that needs to be discussed.

The goal of this chapter is to demonstrate how an operating systems concept can be connected to the Shards system at a high level. As such, it is not focused on the details of the filters themselves but on the wider Shards environment in which they will operate. Filters have already been introduced in detail and their application will be developed further in the next chapter, but the way in which design issues are packaged for them to manipulate is worth discussing. This includes the information that can be expected to be in the application load and how this flows through the filters towards application source code modification and operating system generation.

8.1 Selecting an Operating System Concern

Everything should be made as simple as possible, but not simpler.

Albert Einstein

It is convenient that this foundation experiment should be as simple as possible. The primary reason for this is because it aims to be a demonstration of the Shards system. Introducing too much or too specific material could confuse the issue under a mass of implementation detail. This is also the manner in which filter development should occur. The initial filter family should aim for a high level and abstract implementation which clearly demonstrates the domain. This becomes the foundation for more specialised filters which either extend or reuse the base model in constructing a solution to incorporate some additional functionality. This approach allows operating system practitioners to follow the evolution of a component from a small number of base solutions to a potentially much larger pool of specialised variations. It also allows a new filter construction effort to decide which type and level of specialisation best serves as a starting point.

Abstract models provide a common foundation to build from. They encourage new filter development efforts to consider if they can be expressed as a variant of an existing approach.

This process gathers similar variants into a single filter family. It also allows examination of the filter families to provide an overview of how many unique system concepts are recognised and addressed in the Shards system. Even if this is only as placeholders for future extension into practically useful filters. Being able to say there are N fundamental approaches to an issue and describing them would be a novel addition to the theory of operating systems even if the full implementation of the filters was not complete.

The ability to express a significant functional concern in a simple manner is not common in software and even less so in operating systems. The Shards system helps make this possible by concealing much of the complexity in the creation of subfilters, specialised variants and the operation of the filters. Ideally naming a filter family is enough for the general concept to be expressed.

This demonstration will select a reasonably fundamental component, in this case the resource of computer memory. Examining this component closely will show that it has complex behaviours which offer potential advantages if they can be harnessed. The system being constructed will also interact with both the hardware and how it has been integrated. The design challenge is the degree to which the aims of the system and the strengths of the hardware can be brought into alignment. In the case of customised systems where the system design will have specific demands and architecture there is unlikely to be a universally correct answer as how to best structure this component as part of the system.

These issues will be considered in the design of any operating system. The primary difference is that memory behaviours will tend to be focused on optimal use of the component because the system is not expected to have specific demands other than efficient performance in the general case. The typical ad hoc design process is also unlikely to capture the design decisions that shaped how the memory component was constructed, and what potential advantages were not exploited. A functional and efficient component is the focus and not the broad range of possible implementations. Many details will also be managed by the operating system and not exposed to the higher levels of the operating system or applications.

The Shards system is focused on the creation of filters which will package this resource for use in the construction of customised systems. This approach puts a much higher value on explicitly capturing potential variations and specific design decisions and much less value on a specific implementation. Each practical design approach is likely to find some potential system

configurations in which it is the optimal selection. The complete set of filters will capture all productive ways the resource can be used. Each filter will iteratively evolve to maximise the approach it represents and the possible system configurations in which it can be used.

The development of mature filter families for all permutations of underlying resources, structures built to harness them and interactions with custom system design possibilities is a monumental undertaking. This is the reason why existing system construction tends to focus on a functional implementation rather than the broadest range of possibilities. Fortunately the Shards approach does not have to be complete before it can be productive in a limited role for a specific system. If it can provide a process whereby it is practical and in fact easier to reuse operating system components than write a custom one then the system can expand and evolve over time.

This is challenging because a custom built component can be expected to be optimally well suited for the environment for which it is created. Initially progress must hinge on the opportunities for pure system research and the longer term advantages of reusability. The field of system design will gain if practical reuse can lead to evolutionary extension and refinement of the components to the extent that construction of "from scratch" solutions will become increasingly less required to meet project needs. The long term ambition is that this will decrease the cost and complexity of operating system construction such that the domain for which novel operating systems are economically viable will grow. It is believed that this cannot occur without a reasonably practical framework as expressed in the quote below. Thus the development of this initial framework is the focus of the thesis.

> It's fairly clear that one cannot code from the ground up in bazaar style. One can test, debug and improve in bazaar style, but it would be very hard to originate a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with. When you start community-building, what you need to be able to present is a plausible promise. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.
>
> - *Eric S. Raymond* [Raymond, 1999]

8.2 Memory Management

One of the fundamental aspects of computation is memory, i.e. a location in which to hold data that is to be processed by the CPU. In a certain way the memory, since it connects the individual actions of the CPU into a process sequence, could even be considered the more important of the two resources. As processor speed, application size and complexity of software systems have increased the demands upon memory have grown apace.

The first memory systems were extremely intricate from the point of view of hardware, and exceedingly primitive from the point of view of software. This can be seen in one of the earliest programs, designed by von Neumann [Knuth, 1970] in 1945, in which the program had to consider the complex timing and tiny capacity of the delay line[1] memory system to maximise performance. There was no system software involvement in the process, memory did not have to be acquired (since there was only one program) and addressing was manual and direct with the programming giving specific orders as to which memory cell should be involved in what operation. Of course, this was also fairly tedious from a programmer's point of view, especially as programs became larger and memory technology less exotic.

There were distinct advantages to standardising this common process. This allowed the programmer to more conveniently name and define memory locations. The compiler could do the work of converting mnemonic names to actual locations. It could even automatically detect some common errors such as type violations, which indicates the programmer was trying to put the wrong type of data in the reserved space, and out of bounds memory access. Since memory errors can be hard to debug, as the actual error and its result can be quite disconnected, this was an eminently practical advance.

It was also a required step in enabling the growth of computer systems. Since languages and systems required other elements to be co-resident in memory, such as libraries, kernel code and even other programs, the programmer could not be allowed free access to the entirety of memory. This was especially important in dynamic memory cases where programs could claim memory during their run and then manipulate it freely. As a result the operating system presented, and mandated, the use of memory allocation functions. For the Unix application development [Lions, 1996] this took the form of the *malloc()* function call and its logical opposite *free()*. In reality these are the C library functions which are the common programming interface on the Unix

[1] A memory system in which pulses through the medium of mercury were used to store data.

platform although these will be mapped to system calls by the compiler in producing the binary executable.

8.3 Evolutionary Development

This section will be written as if we are a Shards developer looking to construct a component for which there is no existing material to reuse. At this point in the nascent Shards system, it is very easy to find areas for which this is also the literal truth. However, the design sequence would be quite similar when creating a novel component in a more fleshed out version of the Shards system.

8.3.1 Base Case

When starting development it is desirable to be able to focus on only the area of interest. Any additional work required in areas that are not the focus of development represents a distraction and a dilution of developmental effort. This means that any additional effort required to make the Shards system operate or the operating system build is undesirable. This is a strength of the Shards system in that it allows the system to be constructed even if development of the component being considered is incomplete or indeed even unstarted. This allows the developer to examine the environment in which the component will exist, develop iteratively and test the behaviour from the very first filters being written.

Using the example of memory management, this would be simulated by not scheduling any filters that address this area. This means the application load will not be parsed or examined and any memory operations implicit in the source will be presented directly to the compiler. At the same time the underlying base system will have a memory management component that does not depend on adaptation of, or tuning to, the application load. The applications will be compiled and run on the operating system in a manner almost exactly like that of an existing operating system.

The first step in actual implementation is to consider opportunities for reuse. The first resources sought will be a set of filters to convert the application load into a set of data items on the data chain so that it can be manipulated by filters. The filters sought will depend on the language that has been used to construct the applications. It is possible for multiple

programming languages to exist in the application load and have Shards recognise the language and call the appropriate filter set. In practice, these filters are highly likely to reuse existing compiler frameworks which are mature and complex applications which there is little gain in attempting to rewrite. For example, GCC compiles source code, from the multiple languages it supports, into a sequence of internal data structures. This format may be suitable for use or translation into data chain contents but the process is itself a substantial process as the GCC internals are complex and the internal format was not designed to be exported. The existing Shards system uses a simple custom compiler front-end to read both the project file and some C code, but it is far less capable than GCC.

The data chain is architecturally, semantically and syntactically neutral from a Shards' point of view. It holds general purpose data items while making few assumptions about the internal structure or meaning of the data items. A filter looking to manipulate the data chain cannot operate at this level of abstraction and must make assumptions about the form and content of the data items. At the current stage, this is not a concern as the parser and filters have all been written to work together and assume a C language input as well as a Unix programming environment. It is expected that as Shards develops one or more formats for the data chain contents, probably based around the best compiler internal format, will be adapted and become a common base for development. A small number of standard formats saves each filter developer having to write multiple versions of their filter for each format or filters to convert the data chain formats into what their filter expects to find. The data chain format that has the most filters written to use it can be expected to become the de facto standard.

The extent to which a filter family cannot use a fully abstract data chain format indicates the extent to which it makes assumptions about the architecture of the target systems. For example a filter family could determine that certain Unix inspired design decisions will always hold for every possible system. This makes the filter easier to use with other filter families that accept the same design foundation and harder to use with those those that do not. It is reasonable to assume filters will group themselves around such assumptions while other efforts seek to integrate or clearly identify different sets of assumptions into a more universal form. This could be considered as part of the debate and comparison of approaches to system design. It is expected that for many operations they will be largely operating system independent at the processing level but that remains to be proven in practice.

The designer of a new system will consider how their planned system fits into this debate. They will be able to most easily reuse filter families that make no assumptions or make assumptions which are in agreement with the evolving system design. If the potential for filters to be reused exists then the designer will read their documentation, determine if the function assists their project, or will assist their project if they write some translation filters to integrate them, and either use them, cannibalise them as a initial resource for a custom built filter or discard the filter family.

The designer now has either some filters available for reuse (possibly after some internal adaptation) or a framework in which to fill in the gaps where reuse is not possible. The end result is the construction and inclusion of a number of filters which are capable of finding operations related to memory management in the source code and converting them to a more manipulable form. For example it might locate *malloc()* operations in C and *new* operations in C++ or Java and convert them into a language neutral data structure indicating that the source code is requesting dynamically allocated memory at this point. It is expected that most language's dynamic memory allocation functions will be able to be represented by a reasonably small number of memory primitives. At the other end of the chain there will be code that knows how to convert operations back into compilable code. The simplest form would be a filter that simply rebuilds the original source code construct. In other words the language construct is recognised, parsed into a more convenient form, and then rebuilt. This gives the same net result as the most basic case, but is required before other filters can be put in place.

Since the code is being rebuilt from a logical representation it may look very different from the input source code. The output filters will focus on logical similarity and the needs of the compiler so elements of layout are likely to be given low priority. The further the code has been transformed from the original the less human readable it is likely to become. For most Shards processes, where this code is just an intermediate form for communicating with the compiler this is not an issue. Shards processes which seek to generate human readable output, such as filters focused only on analysis, are likely to invest far more effort in making their output palatable.

8.3.2 Application Modification Case

The next level of interaction is where a change is going to be made but it can be entirely contained within the application layer. This could involve code constructs that would generate

a call into the operating system layer being reorganised so that they will compile without doing so. This could involve modifying the code so that it calls a user-space library or user-space server (the microkernel model) to give equivalent functionality without the need to perform a system call or even for that system functionality to exist. The most extreme form of this would be a computing system in which there is no operating system, as all functionality that would normally result in a system interaction is instead changed to directly included application code. This raises the possibility of the operating system being reduced or even removed which can allow some interesting design trade-offs.

The Shards system may also choose to continue to use existing system calls but modify the way in which they are called so that the application demands are more in tune with the underlying system implementation. This is a process of adaptation rather than replacement and is ideally handled automatically so that the programmer can focus on writing reusable portable code. An example would be reordering or aggregating memory usage within the application to be more compatible with the behaviour of the memory management system.

In both cases these changes will be converted into additional or modified code placed into the application before it is presented to the compiler. It is expected that modern compilers will already perform a number of optimisation operations. These complement the Shards system in that it works between the application and the operating system while the compiler focuses on optimisations between source code and hardware. The focus and the amount of information they have available differs which allows different opportunities for optimisation.

8.3.3 Practical Case

A more realistic model would include some consideration of the hardware environment. For example we can assume the system will have a boot-loader (for example, GRUB[2]) that will handle the initial start-up. On an Intel based system this would include putting the machine into protected mode which is the non-legacy[3] mode of operations on Intel hardware. This process will also set aside a portion of memory for the operating system and interfaces to the machine's underlying hardware. The remainder of the memory will be allocated for software use and can

[2] http://www.gnu.org/software/grub/

[3] *Legacy* in this case being that the system emulates an IBM PC-XT so that MS-DOS will still boot on it. That modern hardware should still have to suffer such an indignity gives a good idea of the relative dominance, and intransigence, of software over hardware in the modern environment. A major iteration of the PC boot architecture (called UEFI and intended to support secure boot) continues to support this legacy mode.

be considered as a contiguous block of memory bounded by a high and low memory pointer somewhere in physical memory.

This is an example of the sort of functionality the Shards system is unlikely to focus on. The hardware initialisation sequence is a one-off operation per power-up and thus generally limited in how much effect it can have on the overall and continuous performance of the operating system[4]. It is also likely to have a single optimal solution for a given system. Stated another way it is an implementation necessity rather than a question of system strategy. While there may be many complexities within the boot up process, the system created by Shards need only be concerned with the environment that is created for it to operate within. If the initialisation process requires arguments then these will be given by the designer in the project file and be unlikely to be changed by any Shards filter as they represent direct communication from the system author to a final component. The initialisation system may also allow booting multiple systems to represent different modes but this is not a concern of the system being run or the Shards process that generated that system.

The more interesting questions revolve around how memory is used in a modern system. In practice, directly using physical memory is not efficient. Reading and writing to memory is sufficiently slow that the CPU will spend much of its time waiting on memory. To reduce this, the system will have one or more levels of cache. These are much smaller blocks of faster and more expensive memory which are used to hold working copies of data from the memory that is in use. The degree to which the CPU can be serviced with data in cache will have a major influence on the overall performance of the system. Since cache is significantly smaller than memory by many orders of magnitude, managing this resource is complex and its operation is supported directly by the hardware.

In addition to the issue, of speed there are advantages to virtualising the physical memory. This allows each application on the system to believe it has access to an extremely large and unbroken expanse of memory for its own personal use. This is known as virtual memory and is common on all modern multitasking systems. The operating system maps any memory actually used by the program to a physical location in real memory. This separation between real and virtual memory allows a number of other system optimisations to be implemented without the

[4]For a system where bootup time is critical, which is a possible design constraint, the solution is almost certain to be solved in hardware. A battery backed, flash memory or disk file containing the image of the loaded system is far more likely to be beneficial than any attempt at software optimisation. For most systems simply leaving the system running between operations is a practical optimisation.

application program needing to be aware of the details although it does make cache behaviour more complex. One example is having some or all of the memory blocks the program is using being moved to a secondary store, such as disk, to reduce pressure on the generally limited amounts of memory available. This is referred to as paging. Virtual memory also allows multiple applications to operate in the same physical memory without collision or interaction by having multiple virtual memory spaces. The operating system manages the mapping of each program's virtual space into real memory. This is required for multitasking in which several processes are active in memory at once.

The mapping between the application's virtual memory address and the physical memory address will occur very frequently. It is also on the critical path for performance as it may further slow the CPU's access to memory. For this reason many platforms will have hardware assistance to speed up this process. This is referred to as a translation look-aside buffer (TLB) and like the cache has a relatively complex internal behaviour. In modern systems where memory is large, the TLB may not be able to hold the entire mapping in hardware and thus will also perform like a cache of currently active mappings.

A given application running in a Unix [Robbins, 2004] styled operating system is distanced from the details of these mechanisms. While the interaction is important in terms of performance, the ability of the program to control and influence specific behaviour is limited. The operating system will tend to have a fixed strategy and encourage access through a highly generalised API. The application will be written in a general purpose programming language which, if well designed, will try and avoid excessive dependence on operating system details. Even the C programming language that was designed alongside the Unix system generalises access to many of its calls.

A Unix application begins execution from a specified point in its code and is able to assume that the region of memory it occupies is contiguous and has been made exclusively available for its usage. The process divides this region of memory into a variety of sub-regions which may also have hardware enforced constraints. The *code/text* segment contains the binary image of the program and any constants, and thus can be considered read-only and will not vary its size during the execution of the program. The *data* section contains data items predefined by the program. As their size is known in advance, this region will change its contents but will not change in size. The remaining memory area contains the dynamic structures that will change

Listing 8.1: Hello World

```
main() {
        printf("hello world");
}
```

both their value and size. They are arranged at the ends of the region expanding towards the center so that as long as memory (or more correctly address space) exists they will not collide. On one side is the *heap* used for dynamically allocated objects while the other is the *stack* used to record the context of a function call. The stack will extend and contract as execution moves through the program code. This use will not result in fragmentation (the existence of unused gaps between data items) or garbage (blocks of memory that are no longer used) and has little structure to exact overhead. It can use this simple approach because data items are expected to be small, short lived and have an inherent sequence in their creation and consumption. The heap is more directly under the programmer's control and thus does not have this regularity in its behaviour, making it a more interesting subject for study.

8.4 Shards Representation

The programmer has quite a different view of the system. Modern programming languages and practice strongly discourage direct interaction with the underlying mechanisms of execution. This is partly because this capability is fairly specialised but more because such interaction cannot easily be generalised, making the code dependent on the system it was written on. This makes the code fragile in the face of system upgrades and difficult to transfer to new systems. Instead, the language will provide its own abstracted model of a process and the compiler is responsible for mapping this to the actual implementation. This process of mapping is now so familiar that it is taken for granted.

In the case of the C programming language [Kernighan and Ritchie, 1988] the most basic model of a process has a long heritage of its own. It is the traditional first program used by many texts teaching the C language including. This code is shown in listing 8.1.

The Shards system depends on filters doing a limited analysis of the source code to identify these implicit assumptions about system operations. Each set of filters will have a specialised

Listing 8.2: Hello World translated

```
(initialise process)
(data item "hello world")
(load data)
(start process)
(text: printf( )
(reference into data)
(text: ); )
(free process space)
```

domain and ignore elements outside of that domain. This is done on the assumption that they will either be handled by another Shards filter or will use the default handling. The default handling is generally being passed to the compiler as is. This allows the module to focus only on material relevant to its domain.

This ability to focus is also demonstrated by the examples used in this chapter. Since the focus is memory usage, we can ignore or simplify the representation of any code that does not interact with this domain. For this initial chapter we will also simplify the detail presented and just take a high level logical view of how Shards operates. This is needed because real examples can quickly become large and contain a lot of detail requiring background knowledge to follow. A more detailed example will be provided in the next chapter. In the context of memory handling the code in listing 8.1 could be translated in the following fashion shown in listing 8.2.

Expressing this small sample of code in this extended form does not add anything to the functionality of the code itself. What it can do is express language specific implicit operations in a more general and easily manipulated form. This form can be standardised between a family of filters such that various languages can all be converted into the same general form where there is underlying operational compatibility. This standardised form can then be recognised and manipulated by downstream filters which seek to modify or define how the system handles these implicit operations.

There is no assumption that there will be one truly universal interpretation of all existing programming languages. Instead there may be multiple occurrences of groups of filters designed to work together. They will be bound by the way in which they codify real world languages in a defined set of system operations. The development, extension and comparison of these encodings will itself be a worthy subject of research. It is hoped that this process will create a significantly

Listing 8.3: Hello World Representation

```
((process,initialise,))
((memory,data,byte),,"hello world")
((process,map_data,))
((process,start,))
((process,function_call,start))
((memory,reference,anon),0)
((process,function_call,call),printf)
((process,shutdown,))
```

smaller number of encodings than there are programming languages and that this process will indicate which models are fundamentally incompatible. Certainly there is a page containing the hello world program in 350 different programming languages [5]. At least for this trivial example it can be expected that most of these examples will translate into the same set of fundamental operations. Stated another way, there are likely to be fewer fundamental operations than there are language representations of these operations.

In the example, the form above is too informal and loosely structured to actually be machine manipulated. The meaningful elements will be codified into elements on the data chain with a header specifying the content. This will allow a downstream filter to infer the internal structure of the data. It would be possible to use a structured notation like XML for this but because the conversion is from text into a defined internal data form in memory, rather than from text to structured text, there is little benefit to this. The filter that will be constructed as part of this thesis will use the notation like that found in listing 8.3.

The above conversion has included two components with one handling memory and another handling process functions. Since this is a fairly simple function the memory operations contained within it are equally basic. If there was additional source code that the memory and process systems could ignore then the source code would have been included (or possibly referenced) as raw source text but not actually converted. The decision of which memory operations to recognise is based on which are interesting from the point of view of optimisation. For example, CPU registers could be considered to be part of memory and managed. However, in practice there are well established techniques for efficient register allocation integrated into modern compilers. If the author of the filter assumes there are no substantial opportunities for

[5] http://www.roesler-ac.de/wolfram/hello.htm

Listing 8.4: Extended Example

```
main() {
        char* heap;
        char[] data = "hello, world";
        char bss[strlen(data)+1];

        heap = malloc (strlen(data)+1);
        strcpy(heap,data);
        strcpy(bss,data);
        printf(heap);
        printf(bss);
        free(heap);
}
```

improvement on this existing functionality the primitives involved need not be mapped.

The memory use in the code sample above primarily exercises the data section of the Unix memory model. It recognises that the string "hello world" will be the first and only element in the processes data section and is referenced later by the index number zero when using the stack in order to construct a function call to the printf() function. An artificially extended version can be constructed that also uses the heap and is shown in listing 8.4.

This code sample is fairly crude and does not contain the includes or any error checking. It would be converted as shown in listing 8.5. This is an example of the sort of material that would make up the data chain.

This extended example gives us a reasonable number of structural elements that are recognised in the code and encoded in a higher-level more readily manipulable form. In the initial stages of filter development, this process is also productive for determining what operations exist and how they can be encoded. It is also expected that a fundamental operation such as calling a function will have different forms in many languages but can all be mapped to a smaller number of primitive operations. The important goal is to not lose information which may be of use to lower order filters. For this reason mapping to primitives should be separated from any suggestion of implementation so that a filter does not force assumptions on later filters and is thus more reusable.

As an example, there are many different possibilities about what could happen when the *(memory,alloc_heap,)* operation occurs. Different operating systems may have substantial variability in how this is implemented. There are also a number of potential strategies on how

Listing 8.5: Extended Example Representation

```
((process,initialise,))
((memory,data,ptr),heap,NULL)
((memory,data,byte),data,"hello,␣world")
((memory,data,blank),bss,13)
((process,map_data,))
((process,start,))
((memory,alloc_heap,),heap,13)
((process,function_call,start))
((memory,reference,named),heap)
((memory,reference,anon),1)
((process,function_call,call),strcpy)
((process,function_call,start))
((memory,reference,named),bss)
((memory,reference,anon),1)
((process,function_call,call),strcpy)
((process,function_call,start))
((memory,reference,named),heap)
((process,function_call,call),printf)
((process,function_call,start))
((memory,reference,named),bss)
((process,function_call,call),printf)
((memory,free_heap,),heap)
((process,shutdown,))
```

this memory is best managed. The particular approach used will be determined by the filters selected but there is no reason they should need novel front end encoders as the input will be the well standardised programming languages the application was written in. It is possible the application load may contain multiple languages that would require multiple filters but this does not change anything.

Note here that *alloc_heap* already challenges the goal of generality by making various assumptions about the underlying memory model being used. An operating system may choose to use a non-Unix-based memory model that does something very different from what is commonly meant by the use of this term. There are two options in this case. The first is to write a filter that maps the language to a different set of primitive operations. However, the word heap is meaningful to developers, and in the interest of avoiding extra work, it can still be useful as a starting point. The application may write a filter to rename the primitive or just directly map it to an implementation that is not heap-based. This allows developers to realise that what would be heap allocation in another system maps to the new mechanism in the system being considered.

It is expected that as processing continues the operations will be converted into more specialised internal forms. The first stage is likely to be conversion into operations that express a strategy for the component under consideration. For example, a language or system using garbage collection will have a substantial number of internal operations and concerns that are unique to that strategy. There remain substantial opportunities for reuse and extension within the domain of that strategy as most implementations will have internal similarities. This reuse is assisted if the multiple implementations of a strategy have a shared ontology. This cannot be enforced but it is hoped the convenience of reuse will encourage the development of an ontology where the underlying operations are actually found to have significant similarity. It is not expected that a universal ontology will be practical as the complexity required will make it unusable. Instead approaches that are sharing techniques and ways of expressing those techniques will benefit the most from a local ontology that may graduate towards becoming a standard once it matures.

Finally the operations will either be converted into source code for presentation to the compiler or turned directly into object code. It is ideal to have this happen late in the sequence as object code is too bulky for convenient manipulation. There will be a number of filters just prior

to this step that perform a high level version of linking. This involves mapping or connecting the operations used for communication between components and any core engine so it can be compiled into a cohesive whole. This would be extremely hard at the object code level where much of the semantic content needed to connect the components has been discarded. As mentioned the core and probably the OS loader are likely to be pre-integrated components for convenience and efficiency although this is not mandatory. The end result is an operating system that can be loaded onto the target hardware and proceed to process the tasks for which it was designed.

8.5 System Interaction

Such a limited example does not have a deep interaction with the underlying operating system. The vast majority of the work happens entirely in the process's own address space. This could be modified by Shards filters but there are two reasons to avoid doing so. The first is that process behaviour is a mature model with few external interactions so there are fewer questions of strategy and approach. The other is that such changes are better performed in the compiler that will be executed after the Shards process is complete. The compiler is the software responsible for converting code into an executable process with hooks into the operating system as needed for things it cannot resolve within the process space.

The operating system's interaction primarily revolves around program initialisation and possibly dynamic memory allocation. On process start-up, represented above by ((process, initialise,)) the operating system sets up a number of memory regions if a Unix like process model is assumed. Two of these sections have a known size and will not vary during the operation of the program. These are the text segment which contains program code and constant or literal variables (and is thus read only) and the BSS segment which contains global variables (and is thus writable). The Linux operating system uses a process called copy-on-write to avoid allocating memory until it is actually used. Thus the system will use virtual memory to map the text segment to the executable code on disk and the BSS to a memory page that has been zeroed. On a write operation to a variable contained in the BSS, a block of real memory will be allocated to hold the write, the zeros copied to fill up the rest of the block and the mapping adjusted. Once fully initialised there is no need for operating system support for access to either of these regions.

The remaining sections are more dynamic. One region holds the process stack which is a

record of function calls, function local variables and function return values. This region is known as the stack and will grow as functions are called and shrink as they return. The other region holds dynamically allocated memory requested by the program code and is known as the heap. In order to avoid excessive interaction with the kernel, both of these regions are given blocks of memory in which to work. The operating system is called only when the regions are exhausted and the process requests that they be extended. The process, or the malloc library on its behalf, may also request that the operating system shrink its memory region when dynamically allocated memory is no longer needed. This may make more memory available for other processes [6]. In Linux there is an additional optimisation in that larger memory allocations are directly requested from the operating system. This is because the heap has a memory management strategy of its own determined by the compiler and managed within the process. These strategies tend to have some amount of unusable space known as fragmentation which large memory allocations can increase.

As a simple example we can consider the null case where we are designing a system that will not have an operating system. In such a case the ((process,initialise,)) primitive is still valid in that there must be something to set up the process. However, what it maps to is going to be very different. In a Linux like system it will map to an operating system call that sets up the process memory regions. In the null case there is no operating system to call. One possibility is that initialisation will be replaced with a small piece of bootstrap code. This code will copy a process image from the permanent store and into memory and then jump to the start of the code. This bootstrap could be triggered by the computer starting up but there are also other design possibilities. It will also be clear that this is analogous with how an OS like Linux starts up when the machine is booted. The end result is that we do not need an operating system to get the process started.

The implementation for this example will require two things. The first is that the filter must be able to provide the bootstrap code. This is complex in its own right, and architecture dependent, so the filter will look at environment variables to determine the target platform and place the appropriate code in position. The second step is that a late order filter is going to have to select (or construct a build script that selects) modified system libraries for the compiler

[6] In practice the operating systems is unlikely to shrink the process space on the assumption it may be requested again at some point and allocating new memory regions is expensive. Instead the virtual memory system will be relied upon to deal with allocated but unused memory

to link against. The compiler uses these to connect the compiled code with the required system calls. This will generally be modified when porting to a new architecture. In this case, the memory management system calls will be replaced with code that will perform similar operations directly. If memory related operating system calls are made directly by the application code then the Shards filter could re-map them to supplied libraries that provide the same functionality without requiring operating system support. Where the memory operations are managed within the process image, the Shards system would modify the libraries used by the compiler for its own process memory management and make sure these are used when building the applications. In other words, when the process calls *sbrk()* to grow the heap, the Shards system will have supplied custom libraries in order to intercept this call. The other alternative would be to remap all memory use in the application code into explicit library calls, which would require a lot more work.

Virtual memory is an essential part of modern operating systems. However in this experiment it becomes simply another design option. Virtual memory could be implemented without an external operating system. The trigger for a virtual memory event actually originates in the hardware when it realises that a virtual address does not map to physical memory. This hardware event could be captured and redirected to a block of code in the process being constructed which would handle it. Assuming an embedded style device is being created, running a single application and focusing on performance by choosing not to implement virtual memory may be optimal. Instead the bootstrap code allocates all the physical memory available to the single process. The compiler is informed how much physical memory exists on the target system which is part of the environment variables from the project file so that the addresses it uses will fit within the physical memory available. The process may then run without a risk of virtual memory events or any of the overhead associated with virtual memory.

The process will run unaware that there is no operating system supporting the memory management. Some of the differences will be that if the process calls the system call *sbrk()* or its equivalent, to increase memory allocation for more heap space the operation will automatically fail. All physical memory has already been allocated and there is no more to give. This will lead to the system crashing or restarting but it could be argued that the problem is that the physical hardware is underspecified for the given task. This can be solved by implementing swapping or memory overlays (where programs are broken into regions that can be written to disk) in the

system libraries. Alternately the complexity and performance cost of these solutions can also be solved by restricting the task being computed or providing more machine resources.

The process will manage memory using its own internal algorithms as before. The operation ((memory,alloc_heap,),heap,13) which requests thirteen bytes from the heap can be resolved internally as long as there is space available. Another design option occurs if the compiler asks the operating system for assistance when a large block of memory is allocated to avoid fragmentation in the heap. The simplest answer is to use parameters to discourage this and modify the system interface so that the call used (*mmap()* on Linux systems) returns a message indicating that no memory exists. A more complex solution could have saved some physical memory for such allocations and would provide code in the system library (which runs as part of the process) to manage the allocation of large objects to this region. Alternatively, the allocation of large objects could be seen as an opportunity to have code use secondary storage such as a disk, if available, to store the data. This would trade off access performance on these objects against the demand for physical RAM. The correct solution will revolve around the design intent. Since the system is being customised to the task, the designer may know that there are large memory objects but they are rarely accessed and decide to use secondary storage.

In short, even with a simple example like this, there are many design possibilities. These become opportunities for optimisation if we know in advance the behaviour of the application and the goal of the system. The changes required to make the system function, such as adding bootstrap code, remapping operating system calls or constructing / selecting custom system interface libraries, are expressed as Shards filters. This allows someone, seeing which filters are used, to understand the system's construction, allows alternate solutions to be selected by modifying the filter selection, and allows the potential for implemented filters to be reused or extended in later versions or other systems. For example, the filters that append bootstrap code to a process would be useful for a variety of tasks including the construction of embedded systems or new operating systems.

8.6 System Generation

The final filters in a sequence will be concerned with generating output to represent the processing they have performed. This could be simply data if the Shards process is being run to provide analysis for the system designer. It could also be output used as an intermediate step in

the Shards process which would represent filters cooperating and communicating through either files or the data chain. The end result sought is to have filters that can generate modules in source code form that can be used as part or the entirety of the operating system construction process.

The first step is to integrate any changes in the source code of the application load. This will generally be performed by using a filter to convert the representation of the source file in the data chain back into a source code file. This file temporarily replaces the original and is not expected to be human readable. The build process for the application is triggered, the binary built, and the original file returned to its original place.

The designer has a variety of choices in how deeply Shards will be involved in the construction of the operating system. On the principle of fall through, the Shards system attempts to work within the existing process so that construction of the operating system may be assisted by Shards without being dependent on it. In other words it will be possible to compile applications and system as normal if desired.

The lightest interaction is the Shards system being used purely as an analysis framework. In this case it would generate human readable output indicating potential performance concerns or optimisation possibilities for designer attention. The next step would be generating the same information as a project header file so that code within the operating system could access and operate on the basis of the information generated by Shards. This information could also be used by the build process to enable conditional selection of source code for building the system. As an example, if the operating system had two memory manager implementations (and there are many available in the domain) it might use Shards analysis to determine which one is built into the operating system.

The deepest interaction is when the Shards process is run again on the operating system code itself. In this case, the output of analysis provides input to the Shards system as if it were designer provided environment variables. The operating system source code is then read into the data chain in the place of the application load. The selection of which source code to process will be determined either by developer hints or which source files export and support a particular system primitive. Filters are scheduled to examine or modify the operating system source code (and build process) based on the results of the first analysis run on the application load. At the conclusion this source code is written to disk in the same manner as the application

case and compilation of the operating system proceeds. This could also include the generation of entirely new source files and modification of the build process to include them.

In practice it is expected that only a small proportion of operating system code will be changed even in the largest Shards implementation. In most cases providing tuning information and selection between alternative source code implementations of system operations should be sufficient. This approach provides a lower dependency between Shards and the operating system build and enables easier isolation of errors. The capacity for custom code generation does exist and potentially, bounded only by the substantial complexity of implementation, would allow the operating system to be tightly tuned to the design goals expressed by the designer (as environment variables and filter selection) or determined by analysis of the application load.

8.7 Summary

This section covers only a very small section of the breadth and depth that can be found within operating system implementations. The goal was to demonstrate the approach which a Shards user will adopt in seeking to limit the scope that must be considered and captured. The primary driver is in using application code to find interactions that extend outside the process and involve the operating system. These become the hooks for possible change and optimisation from the default handling. Selecting a single subsystem, such as memory, and determining which requests it will be satisfying, allows a boundary to be drawn and much of the application code and other systems to be omitted from consideration. Within a subsystem many mechanisms will not have interesting optimisation possibilities (such as the boot process by virtue of being a one-off operation and a tiny portion of the total run-time) and may also be omitted. What remains are the elements most interesting from a operating system design sense and thus most desirable to capture.

Chapter 9

Extended Example

This chapter will focus on a more detailed example of the Shards system and the process being applied. It will focus on a single aspect of operating systems to keep the scope manageable. This will be the subject of memory management which was introduced in the previous chapter. It will also continue to be somewhat abstracted in the interests of brevity and being able to focus on the usage rather than implementation details.

The technical content is not intended to be authoritative in terms of a analysis of memory management issues or solutions. The amount of detail presented was determined by the needs of the example and not on the accuracy and completeness of the outcome. The optimisation possibilities were also selected on the basis that they provided a compact technical challenge to demonstrate the Shards approach. A practical analysis of the limits of current mechanisms, the creation and implementation of innovative new solutions as well as documenting and proving the advance will remain challenging and creative processes that cannot be automated and thus have little cross-over with the Shards process. The Shards process is a method for capturing the output of that design process such that it can be productively understood and reused in the construction of future systems.

As before, the examples are going to use the Unix environment, and more specifically Linux, as a foundation. The Shards system is not Unix specific and could be adapted to other existing systems or entirely novel systems. It is not the purpose of this thesis to invent a new system and Unix remains a well understood and practical foundation for operating system discussions.

9.1 Software Environment

From a C application programmer's point of view, dynamic memory allocation is conveniently simple. The function call *malloc(N)* provided by the standard library is a request for N bytes of allocated memory. The system will return a pointer to the requested region of memory or it will return a null pointer indicating the request could not be satisfied. It is expected that the call will succeed, barring memory actually being totally exhausted, which is something the operating system cannot be expected to resolve automatically. There are a variety of other calls (e.g. *calloc(), realloc()*) which represent variant forms of malloc with a behaviour that may be more convenient for a given task. The end result of all of them is that a block of memory will be requested from the system. The call *free()* indicates that the programmer has finished with the block of memory and it can be reclaimed by the system.

As introduced in the previous chapter, a great deal of an application's memory demands are managed without operating system intervention. The application will request a block of memory on process start-up and use this for storing dynamic memory allocations. This area is generally referred to as the heap. It will run its own memory management algorithms to effectively use this space. For example, the GNU C++ compiler has used a library called DLMalloc written by Doug Lea [Lea] for this purpose. The operation of this mechanism is effectively invisible to the operating system as it is entirely contained within the process image. It is also likely to be optimised for efficient storage of small memory objects as many dynamic allocations may be small while the operating system's own memory management tends to deal more with larger objects.

In a Linux environment the process will interact with the wider operating system for assistance with memory management in two ways. The first is that it will request the operating system to extend the space allocated for the heap when its current size is unable to satisfy a demand for a new memory allocation. This may not mean the memory allocated to the process is full as memory management algorithms have to deal with internal fragmentation. Memory fragmentation occurs when the memory allocation algorithm produces some small or unusable blocks as a side effect of its operation. There is generally a trade off to be made between speed of allocation and fragmentation so this is another design decision. In any case, if the internal memory cannot satisfy the request then the process will ask the wider system for more memory to manage.

The process will use the Unix system calls *brk()* or *sbrk()* to request additional size. The name is a reference to the *break point* that separated the heap from the stack. Traditionally the heap used to occupy the low end of available memory and grow upwards while the stack occupied the high end of available memory and grew downwards. So this system call historically meant to adjust the line of division to give more of the memory region to the heap. It is now archaic as on a modern system both the stack and the heap have their own independent virtual memory mapping. The first function takes as its argument an address to set as the new heap endpoint while the second takes the change in size needed. Both of these calls can be used to shrink the heap as well and the process may do so. In practice, not returning the memory and allowing the operating system to swap out allocated but unused memory is easier and more efficient for the process and thus is a viable design choice.

A process compiled by GCC and running on Linux (and probably for other compilers and systems) has another option. In order to simplify the management of its memory and reduce fragmentation it will handle large allocations differently. Rather than allocating these large objects on the heap the process will call the operating system memory allocator directly to satisfy that allocation. This tends to be efficient as the operating system allocator will be tuned towards larger allocations. It also relieves the process' internal allocator from managing the data object. There is an overhead in setting up this allocation which is the reason it is reserved for larger objects. The user can tune the behaviour of this mechanism using the *mallopt()* library call which is able to set what size will trigger a system allocation request and how many may be active at one time.

The operating system call used for requesting a new memory region is *mmap()*. This can be called by the application programmer directly as well as being used by the process to request memory for a large allocation. It is also used by the operating system itself when it needs a memory region such as when setting up a new process. One use of the function is to map a file into a region of memory. For example, the executable code for an application resides on disk but is mapped into virtual memory as one of the process regions. The dynamic usage is called an anonymous mapping in that it opens a region of memory that does not map to a file on disk. The arguments it takes are a size, a number of flags that configure it as a private writable memory allocation and an optional starting address. In general, most anonymous mappings are not concerned with where they exist in memory and the optional argument is not used, allowing

the operating system to freely place the memory block. The function call *munmap()* allows memory allocations to be returned to the system.

The operating system runs its own memory management schemes in order to service memory requests. This may be from processes starting up or processes calling *sbrk()* or *mmap()* but it also has to deal with a variety of internal demands. Since the focus is on application driven requests some of the more specialised and internal considerations will not be covered. There is an immense variety of strategies [Wilson et al., 1995] that may be applied to balancing access speed against maximal use of space (including avoiding fragmentation) against the processor time and memory overhead of running the system. In order to keep things simple this chapter will use the strategy employed by the Linux system as a starting point.

Individually tracking every byte of memory available in the system would require a great deal of overhead since data about memory must itself be stored in memory. For this reason, both operating systems and the hardware deal with larger memory blocks referred to as page frames. On the 80x86 platform (the common PC from Intel) the hardware page frame is four kilobytes (4kB or 4096 bytes) and Linux also adopts this size when implemented on that hardware platform. Each page frame has an entry in a global array which records information about that page such as whether it is in use.

Memory inside the computer may not actually be uniform in its structure. It may be broken into different hardware regions which have different access times or internal properties. For example, on an Intel system certain memory may be used for a hardware driven memory transfer known as direct memory access (DMA) which is efficient because it does not require the CPU to invest effort in managing the transfer. The underlying hardware that performs this operation cannot span the full range of memory addresses leading to there being some areas in which DMA will work and some in which it will not. The Linux system divides memory areas with identical properties into zones and organises zones with similar access behaviour into nodes. The specifics of the division are hardware dependent but the essential point is that a zone within a node is a block of memory with uniform behaviour.

The internal call used within the operating system for requesting memory is *alloc_pages()*. It takes the order of the memory size requested (giving N is a request for 2^N bytes) and an argument consisting of flags to inform the allocator of desired behaviour. This will enable the zone allocator to determine the correct zone to satisfy the request. It will automatically try to

balance requests so that they are equally balanced amongst the available appropriate zones and is responsible for calling the operating system facilities for dealing with low memory situations when required. The arguments also contain directions on how to handle the memory request if low memory is an issue. For example, the memory allocation requests from the application level are given a number of flags which effectively state that the process is willing to wait if memory is not immediately available. This gives the operating system freedom to block the process while it attempts to free memory to satisfy the request. Memory is returned to the system when released by the application with the command *free_pages()*.

Each zone runs its own memory management scheme. There is a cache of single pages referred to as the Per-CPU page frame cache that enables requests for single pages (which thus cannot cause fragmentation) to be supplied quickly. This is separated into cold (available for use) and hot (recently used and thus may be in the hardware cache) pages. Requests for an allocation region larger than one page frame will be satisfied by a buddy system allocator [Gorman, 2004]. This allocator divides physical memory into order of 2 (e.g. 1, 2, 4, 8) page frame groups up-to four megabytes. Each free region is kept on a list matching its size. The allocator is fast and low overhead because it only provides memory regions in these specific sizes. For example if the request is for five page frames then it will return an eight page frame block because this is fast to allocate even though it does mean some space will be wasted. It will not split the block into five frames used and three available. If there are no eight page frame blocks available then it will split a larger block in half (possibly recursively) in order to create the desired memory region and at least one free block of the same size. When the region is freed, it only checks to see if its neighboring block (its buddy created when a larger block was split to create it) is free in order to combine into a larger free block (once again possibly recursively). This combination of only dealing with fixed sizes and one boundary is what allows the algorithm to be fast, computationally cheap and not require much record keeping.

At this point, a block of physical memory has been found and can be given to the process. The application's usage of *malloc()* may trigger an allocation request either to store the object (if it is large) or to grow the process heap. These will map to the library calls *mmap()* and *sbrk()* respectively. The C library will map these calls to the operating system interface which will call *alloc_pages()* with flags indicating it is a process request. The zone allocator will determine the appropriate zone from this call and the memory management system within the zone will

Listing 9.1: Initial Parsing

```
((memory,data,ptr),X)
((memory,alloc_heap,),X,N)
((memory,reference,named),X)
((memory,free_heap,),X)
```

determine the memory to use and mark it as no longer available. In short, what looks very simple to the programmer can cause an entire sequence of operations and the interaction of multiple subsystems before it can be satisfied.

9.2 Initial Shards modelling

The input process was discussed in the previous chapter. The essence of the process is that the designer places a number of filters in the project file to indicate source files of interest. For example the filter *parse_c* with a pathname as an argument will parse either a single file or a directory full of files. The parsing will be performed by filters dedicated to that task, reuse of existing compiler front ends, or some combination of the two. The result will be a data chain representing the contents of the source files. It will also be quite large but only a simple form will be considered here. The filter will only respond to items relative to its area of interest which is in this case dynamic memory operations. This allows the omission of all operations not related to these operations from the discussion as the filter is expected to just skip them. The result is the sequence indicated in listing 9.1.

In the current analysis the source code files are being treated as if they are simply text files. The operations are translated directly from the source code and the operations arranged in the order they are encountered. This does not reflect the actual path of execution which is substantially more complex. This is sufficient for identifying specific calls but will limit the detail on analysis since the reference to a particular memory may depend on program flow and even the creation of the memory area itself may be conditional.

This level of analysis cannot address dynamic conditions but it can identify events. Thus while an allocation of memory may be conditional, and the reference to it may be uncertain, a filter can still identify that the call exists in the source code and consider potential for manipulation. This would be triggered either by the designer setting a relevant environment variable,

scheduling a filter that optimises memory or a combination of the two.

The designer is able to consider the information that will be gathered and the context of how the system will support the operations in order to look for optimisation possibilities. In this relatively simple case, and for such a primitive operation, it is unlikely that such a possibility will be considered. However only the specific design goals of the system project, and the perception of the designers, can determine whether possibilities exist and are value enough to be acted upon.

One of the main advantages of the Shards system is that if specialised handling is desired then it can be done on a case by case basis, without incurring a run-time performance penalty. For example, if the designer wanted to support a different mechanism depending on whether the memory allocation was above or below the median allocation size of the application this would be difficult to implement as an operating system enhancement. It would require metrics to be kept on what the current median size is and a decision in the process of allocation as to which mechanism to use based on allocation size. This process would exact a cost in performance that would be applied to every memory allocation operation, even where the enhancement is not required or beneficial. A Shards system could determine the size of allocations from the source code and modify the actual function calls to determine which mechanism is used. Since these changes are all done prior to compilation of the operating system there is no run-time penalty.

In this example the designer considers the possibility of implementing their own memory management within the process' memory image. This would be implemented by modifying the process so it requests additional memory as part of the initialisation phase and then satisfies dynamic memory allocation requests directly out of that pool. In effect what would have been system calls are converted into internal calls on a memory management library that has been compiled into the program image. This would be a potential advantage in speed of allocation, and allow more control in terms of ordering the use of memory, but it requires extra work to implement and may mean that memory is ultimately used less efficiently system wide if each process maintains its own private memory area. The determination of whether to implement a particular optimisation is part of the design process and ultimately determined by the project goals and resources available.

It may prove simpler to work with the API provided by the operating system as this reduces the amount of effort required. The Unix API is focused on reducing the complexity of software

development which includes limiting the number of API calls to avoid repetition, the number of arguments required to each system call and the prerequisites needed to use the system call. This creates an easy to use API but also reduces the possibility of tailoring the behaviour of the calls. This is a reasonable trade-off for a general operating system which cannot base its design on the unknown optimisation demands of the multitude of client applications it will support. The Shards system which works with custom systems, specific design goals and known application loads has the potential to derive a great deal more information which could be used.

A reasonably simple way to use additional context information would be to short-cut the memory handling hierarchy. As was described in Section 9.1, memory handling requests potentially flow from language, to compiler provided manager, to the general Unix memory manager, the underlying system specific managers and finally to the hardware. The designer could construct a filter that converted some or all malloc calls to calls that went directly to one of these layers of memory handling. For example, the malloc calls could be adapted to directly access the underlying Linux memory managers for some or all calls (though it would only really make sense for a very large allocation). This would enable the process to directly interact with the behaviour of the underlying system memory managers even though the application source code was written to be system neutral. The advantage is increased control over which system mechanism will be used to satisfy the request at the cost of complexity. The fact that the process is now specifically tied to the system is of little concern as this is assumed to be the case for a custom system in which a change of operating system or hardware will only be performed as part of an upgrade project.

Finally, the Shards process could gather statistics summarising what it can discern about memory usage. This could either be printed out in human readable form for consideration by the system designer, used to drive some change in how the operating system is constructed, used as arguments to tuning the system (either through modifying the build process or using a provided system call like *mallopt()*, or any combination of these. When creating an optimised custom system information is a valuable commodity.

The designer considers the application load being used, the information gathered from analysis tools or Shards analysis filters against the goals of the project and the resources available. The major question will be whether there are substantial optimisation possibilities that justify spending the resources to attempt to take advantage of them. This calculation is likely to be

modified if there are existing filters that provide the facility needed and are reusable as this means the resource cost drops dramatically compared to creating new filters or code. For the purposes of this example, the designer feels the correct course of action is to leave the code as it is. No filters will manipulate the process by which memory is provided to the application process and the default system mechanisms will be used. The calls shown above will be converted back into C code and presented to the application build process. If no filter has indicated a modification to the given file then this step could be skipped entirely. However the designer decides to continue looking. There are more layers of memory management which may provide new opportunities. Specifically the designer will consider the hardware on which the system is going to run. As with an API the operating system will generally abstract the specifics of the underlying hardware. This is beneficial in the general case, but for a custom system it may be concealing optimisation possibilities.

9.3 Hardware Environment

As has been shown above, the programmer's interface to the dynamic memory system is simple and convenient to use. Regardless of the purpose to which the memory will be put, only a single command needs to be used. Other languages are likely to have similar operations or operations that can be mapped to the basic semantics of requesting a linear array of bytes and then managing it within the language. This clean interface gives little insight into the multiple software subsystems that must interact to satisfy this request. It also gives the operating system little information from which to determine the planned usage for the block of memory. This sequence completes with the lowest level memory allocator finding a correctly sized block of free memory whose address can be returned to the process.

The programmer's interface also contains another assumption. This is that every byte contained within the newly allocated data structure is equivalent. The truth of this assumption is a function of the tolerance to sub-optimal performance which the process will accept. If performance is only a moderate concern then modern memory can be considered to have uniform access latency. However, as the demands on performance increase, any variation in latency becomes more important. At the most demanding levels, where optimisation is the primary goal, it becomes clear that the assumption is entirely false. Different bytes within the allocated memory can indeed exhibit differences in relative latency and in fact, this difference might not

even be regular.

The reason is that the abstraction provided by the memory allocator hides the fact that physical memory is itself a system constructed from other components. These components, and the interaction between them, can affect the behaviour of the memory access. The most dramatic of these is of course the existence of a high speed cache. For modern computers main memory access latency represents a serious reduction in processing power. The hardware provided memory cache which can respond with a much lower latency is an essential tool in concealing this restriction. However because the cache is much smaller (as it is significantly more expensive) than main memory it can never guarantee that it has the requested data available. There is a probability of it failing to contain the required data and the transaction having to accept the latency of accessing main memory. This is known as a *cache miss*. As a result the latency of a given byte will vary, dramatically, depending on its status within the memory hierarchy.

> Traditional multiprocessor operating systems, including those commercially available today (e.g. IBM's AIX, Sun's Solaris, SGI's IRIX, HP's HP/UX), all evolved from designs when multiprocessors were generally small, memory latency relative to processor speeds was comparatively low, memory sharing costs were low, memory access costs were uniform, and cache lines were small." [Gamsa et al., 1999]

The more performance demanding a process becomes, the more optimisation possibilities will be worth considering. It has even been identified [Cuppu et al., 1999] that the physical architecture of the memory hardware has variations within its internal accesses. Indeed advances in the capacity and speed of memory are generally dependent on more complex mechanisms within the memory chip. A side effect of these mechanisms is an increase in variability of access behaviours. The medium of communication between these memory components is not neutral either, with the modern system bus being both a complex and shared system. System bus contention, which becomes more of a factor as the speed and number of the system components increases, will also affect the behaviour of memory accesses in a non-uniform manner.

Table 9.1 provides some statistics on a reasonably modern production memory system. As can be seen, cache performance is now so essential to overall system performance that three levels of cache exist. Effectively the cache is, itself, cached. Level 1 cache represents the memory closest to the processor, almost certainly on the same chip and thus can offer the lowest number

	Level 1	Level 2	Level 3
Cache Lines	256 x 64 bytes	2048 x 128 bytes	12,288 x 128 bytes
Set Associativity	4-way	8-way	12-way
Policy	N.R.U	N.R.U	N.R.U
Method	Write-through	Write-back	Write-back
Latency	1 cycle	5-11 cycles	12-18 cycles

Table 9.1: Itanium Memory Hierarchy

of CPU cycles required for access. There's also a level 1 cache dedicated to CPU instructions, known as the I-cache, but this chapter will not be considering that subsystem. Level 3 cache is the slowest, an access representing 12-18 processor cycles as indicated, but as can be seen it is much larger and has the highest associativity. All of these caches are dramatically faster than the latency involved in accessing main memory.

It is also worth mentioning, primarily for completeness, that this memory hierarchy can also be extended into the core of the CPU itself. Programmable registers, which hold data actively involved in computation, are actually the fastest, most expensive and most in demand data storage. The number of registers, and the efficiency of their use, has a dramatic potential to affect the performance of the software being run. The primary reason this will not be considered in this chapter is because register allocation can be effectively automated through the technique of register colouring [Chaitin, 1982]. It also has such a high rate of change that software intervention specifically aimed at influencing this mechanism is unlikely to return substantial optimisation possibilities.

The terminology used in this chapter for describing memory will require some explanation [Patterson and Hennessy, 1998]. The first element is how the cache is mapped to main memory since clearly the cache is significantly smaller than the main memory. The solution is to effectively divide the main memory up into cache sized blocks. The memory within each of these blocks has a one to one mapping with the cache called *direct mapped* in cache terminology. Of course this means that each addressable block in the cache is responsible for (memory size / cache size) regions of memory only one of which can be resident in the cache at any given time.

An addressable block in the cache is known as a cache line. The ideal size for these blocks is influenced by the principle of spatial locality. The theory is that when a specific byte in memory is addressed there is a high probability of sequential access on the nearby bytes. Since memory communication delays are primarily involved with latency, rather than transmission delay, it

makes sense to get both the byte and some amount of the data around it. For this reason the cache deals in blocks known as 'cache lines'. If a byte is accessed then the entire cache line in which it falls is fetched and cached. The minor disadvantage of large cache lines is that each size extension has a progressively lower probability of including data actually related to the targeted byte. The major disadvantage is that the larger the cache lines the fewer lines a cache of a given size can hold. The larger the line the more physical memory it is responsible for as it will manage this size region in each of the memory blocks that maps to this cache line. This makes it more likely that some other process or a later stage of the current one will manipulate one of these other regions. This will cause it to be brought into the cache line and displace the existing content in what is known as a cache *collision*. In the worst case two processes will repeatedly be accessing the same cache line and evicting the others cached data in a process known as *cache contention*. This is mitigated by keeping cache lines small.

Using large cache lines designed to profit from exploiting spatial locality represents a cost in terms of *temporal locality*. This property is an identification of the fact that if a data item is referenced then there is a strong probability it will be accessed again in the near future. Thus it is desirable to leave it in the cache so that it is still available when the next access occurs. Large cache lines require more space within the cache, which means existing data must be flushed to make space. As can be seen the more flushing that occurs the less able the system is to take advantage of temporal locality.

One way to restrain contention over cache slots is to have more flexibility in placement. In the strict direct mapping system each block of memory can only exist in a single line of the cache. This means that if multiple data items in use all resolve to the same cache line there can be a conflict over that line in the cache, even if the remainder of the cache is empty. In short it makes the cache susceptible to *hot spots* of activity. This is reduced if data items can be stored anywhere in the cache, called *fully associative*, which means that the cache will not exhibit pathological behaviour when there is an unfortunate alignment of data.

However, the disadvantage is that fully associative caches need more hardware logic to support the overhead of the more complex algorithm. This represents both additional transistors consumed which reduces storage capacity and extra delay, scaling as a factor of the cache size, over the simpler direct mapped solution. There is no solution that offers the best of both approaches, so the systems answer is to produce a compromise. In essence the cache lines are

organised into sets, with allocation being direct to the set and then fully associative within it. This allows one extremely active line in the cache line set to spill over into its hopefully less active partners and thus evens out hot spots. The Itanium uses this approach, called set associative, with the size of the set (the N in N-way in table 9.1) growing as the tolerance for the cost of the additional logic and delay increases.

The next attribute describes the write policy for the caches. The question here is what to do when a write is made to the cache, because as soon as it receives the new value it is no longer coherent with main memory. In effect the value in memory has become stale, useless at best and a source of error at worst. The simplest solution is *write-through* in which the memory is updated at the same time as the cached value. This solution generates memory traffic for each update which can be both very frequent and have an extremely short lifetime before it is updated again (consider for example a loop iterator). A better solution is write-back in which the memory location is updated when the modified value is evicted from the cache. This reduces memory traffic dramatically and also represents that being removed from the cache ideally represents the cessation of updates, however it is more complex to implement. Thus it is no surprise that the high value L1 cache (which can also be assumed to have a dedicated and fast channel to the L2 cache) implements the simpler scheme.

Finally, there is the eviction policy, which is a logical consequence of having an 1 to N mapping between memory and cache blocks. When a new data item must be cached a decision must be made as to which existing block will be evicted. Ideally the least valuable block will be selected, but this value can only be detected heuristically. The most desirable scheme is LRU or Least Recently Used, in which the block that has been unchanged for the longest is selected for eviction. However this scheme requires additional complexity and state data to be contained in the control circuitry. Thus all of the Itanium caches depend on the simpler NRU or Not recently used in which a bit indicates whether the item data has been accessed since the last sweep over the cache storage. Since there may well be multiple possibilities for eviction unlike LRU the final determination may involve randomly selecting one of the alternatives.

As can be seen from the above, memory is far from uniform in its behaviour. The difference between one access and another has the potential to be variable, and the extent of the variation is sufficiently large as to impact overall system performance. Indeed as systems become faster and more advanced the dependency on more complex support systems grows. These systems have

more complex behaviours and a higher variation between their ideal and worst-case outcomes.

9.3.1 False Sharing

As the importance of achieving a cache hit has grown, along with the capacity of the cache itself, the temptation is to grow the cache line. There is a reasonable probability of a locality hit on the additional data that is loaded along with the desired data item. And such a hit, since it means that a potential cache miss is avoided, represents an appreciable performance advantage. If it fails and the additional data cached is not used then no harm has been done and it will shortly be evicted from the cache.

However, the situation changes when there are multiple CPUs in the system. A multi-CPU environment can now be found in domestic level computing hardware. In such an example the data brought in alongside the cache fill might actually be a part of an active process on one of the other processors. This situation is known as *false sharing* because the cache consistency mechanism within the system hardware (such as MESI [Papamarcos and Patel, 1984]) recognises that the same cache line has been accessed by the caches of two different CPUs. The fact that the two processors may well be accessing entirely different memory items is lost because the granularity of the system is equal to the size of the cache line. As cache memory block sizes grow the probability of this sort of collision increases.

The result is that the cache line in question must be flushed from any caches it exists within. This has the potential to turn a potential cache hit, in the original processor, into an expensive cache miss. Even worse that miss will also exhibit false sharing causing a reciprocal cache invalidation. In the worst case two processors can *ping-pong* the cache line as each attempts to seize it from the other. What should have been a stream of cache hits has become a stream of cache misses for two data items that are completely unrelated outside of their spatial locality in memory.

This is particularly acute in the case of data used as a lock for shared memory regions. This data item will tend to be small, but it will potentially be heavily shared. Each user will be attempting to gain write access, the most restrictive form which demands other caches are flushed, in order to claim access to the region. Once the lock is seized it will often not be held for long as lengthy ownership of a lock will force other processors to wait on release. This design, while eminently logical at the application level, provides something close to a worst case

sequence at the cache level.

This situation can be solved in the compiler, but not optimally [Jeremiassen and Eggers, 1995]. The restriction is that the compiler needs both system information, in terms of the cache size, information on the run-time allocation of its data objects, and an understanding that other processes will reciprocate in order to perform optimally. Without that system information the only safe heuristic to counteract false sharing is to use an entire cache line for the data even if it will contain empty space. This will exact some cost in memory usage and reduced spatial locality since having empty or restricted space in the cache will slightly reduce the chance of a subsequent cache hit. While this is generally a worthwhile trade off in small footprint CPU intensive applications, it will become less optimal as the size of the applications memory use increases. It also lacks information on the focus of the application. If effort is put into optimizing performance for an application that is not performance limited then there is extra complexity for no practical gain.

This pressure is multiplied if the caches in question are not sharing a common communications channel. The existence of this channel is a prerequisite for many of the most efficient cache coherency schemes because it automatically acts to sequence and broadcast update messages. A shared channel is also a scaling constraint however, and thus larger systems tend to have both disjoint memory and disconnected processors. This means that the delay involved in detecting cache invalidation, and in accessing the now freed block, tend to become larger as the system grows.

9.3.2 Memory Page Structure

The cache block is the largest structure the processor deals with directly. However it is not the largest memory structure that exists within the hardware system. The individual cache lines, large as they are relative to a single data item, are far too small to span the large main memory of a modern system, an expanse of memory which while dramatically slower and with far greater access latency provides the many hundreds of megabytes that modern software can consume.

This memory is organised into *pages*, which is the fundamental structure on which the operating system and hardware operate. While there is theoretically nothing stopping these pages from being the same size as cache blocks there are practical limitations that make it sub-optimal. The primary issue is one of scale, as the operating system needs to both index

and assign attributes on a page basis so that it can share memory between processes. Dividing main memory into such small blocks, and then multiplying the resulting number of blocks by the index and attribute memory requirements, would result in substantial amounts of memory being consumed. Thus in practice operating systems have worked with a larger size, traditionally 4Kb but increasing memory sizes have made even that potentially too small. The Itanium processor is designed to work with a range of page sizes in expectation of future growth and can handle sizes from 4Kb to 256Mb.

Memory pages are one of the primary structural elements behind a great number of operating system facilities. This flexibility stems from the fact that it is possible to add a layer of indirection between the program's idea of memory and the true memory. The software version of memory is referred to as the *virtual* mapping. Software depends on the operating system to provide a *virtual to physical* mapping that turns an access on a virtual address (which is all the memory the CPU and operating system could potentially address) into a real memory location. The advantage is that the operating system is able to act as an intermediary on memory access, and that it has some flexibility to manipulate the environment thanks to this layer of indirection. This is an advantage to both the program, that can be written to target a version of memory much simpler than the real thing, and to the operating system which is able to provide many system facilities.

From the previous introduction to the cache it should be clear that there is a fundamental conflict here. The processor is extremely demanding in terms of the rate at which it wants access to data. Invoking the software operating system and allowing it to calculate a mapping before the access to memory can even be initiated represents a crushing reduction in peak performance. This would be the case even with the operating system having previously established a desired mapping as it would in most cases have no interest in intervening in the operation.

The translation table, the data structure at the heart of the mapping process, is too large to be contained within the valuable hardware of the processor. This is especially true because the virtual map is likely to be bigger than the minimum required to span the available physical memory. This allows the operating system to use the entire virtual space for various operating system features (for example swapping regions of currently unused memory to disk) even if only a section is currently backed by physical RAM. The solution is another level of caching based on the assumption that an active process will only be actively working with a small subset of the

total pages in memory (also called the working set [Denning, 1968]). This cache of translations is referred to as the translation look-aside buffer or TLB. Since this is not an application programmer concern it does not have the same fame as the cache and is discussed in detail only in very low level hardware specific resources such as Intel's hardware manuals [Itanium2hw:2002; Itanium2rm:2004] however its importance in performance is substantial;

> TLB coverage has been identified as a potential bottleneck in system performance, with TLB miss handling overheads of 20-40% reported even on single-tasked benchmarks [Chapman et al., 2003]

The Intel manuals quote a figure of 20 processor cycles before the request for a missing translation entry is even initiated to memory. This cost and the resulting load of the translation from memory must be paid before any operations on the actual memory item can even be commenced. The above quote also introduces the term *coverage*, which is a reflection that this cache is a limited resource under contention. The number of pages it can hold determines the amount of memory that can be addressed before TLB misses start to occur. In practice this maximum coverage is further reduced because memory is not solely configured for the convenience of the TLB. There are also other affects such as aliasing [Wiggins, 2003], which represents a physical page having multiple virtual mappings, which can also reduce the effective coverage of the TLB[1].

The Itanium Architecture in reference to the importance of TLB performance devotes a substantial amount of resources to the problem, even to the extent of having a multi-level caching scheme. To give some specific numbers the Itanium-II L1-DTLB [2] has 32 entries which are restricted to handling 4Kb physical pages and do not cache the full access permissions for the page. The L2-DTLB has 128 fully associative, multi-sized and permission complete page mappings. The Itanium also allows up to 64 of the entries (which it calls TCs for Translation Cache) to be filled from software (called TRs, Translation Registers) at which point they are exempt from hardware filling or flushing. Of course each utilised TR represents a decrease in the coverage provided by the TC, and thus an increase in the pressure upon it.

[1] Although the Itanium Architecture, since the cache works solely on physical memory addresses, is not susceptible to many of the aliasing concerns.

[2] Data TLB. The Itanium is a modified Harvard architecture and has independent systems for instruction and data entering the processor. There is a largely separate cache and TLB structure, not covered in this document, devoted to reducing instruction latency.

9.3.3 Non-Uniform Memory

In the previous section on caches the argument that different bytes could have different and variable access latency was advanced. This was due to the interactions of the mechanisms within the hardware on memory. In other words even if memory truly is uniform in its access latency the processor's access mechanism will create variation. At the system level where the physical memory is managed it is possible to identify cases in which the physical memory is itself non-uniform (although variations due to memory chip construction has already been raised).

One obvious example comes from the facility known as *virtual memory*. This mechanism uses the indirection provided by the virtual page mapping and made practical by the TLB to allow a process to access more memory than is physically available on the system. This is convenient from the program's point of view, making it much less likely that a request for more memory will be rejected. It's also convenient for the system because it allows memory to be more highly utilised. It can do this through exploiting the inverse of locality. If a process spends much of its time in a relatively limited amount of memory then there is by definition a large amount of memory in which it spends very little time. If the data contained in this memory could be moved to a lower class of storage, which in most cases means the hard disk, then more memory is available for data actively being processed.

What this means in practice is that when the system is under memory pressure pages can be removed from main memory and copied to secondary storage. The access times on data contained on that page is now determined by the time to copy it from disk and back into memory. For the processor, which considers main memory to be unacceptably tardy, this latency is approaching disastrous[3]. If the data is requested then many millions of processor cycles can pass before the data is ready. As a result the operating system invests substantial effort through heuristics and access statistics to try to balance the gains and costs of virtual memory.

An unexpected example of usefully non-uniform memory came from discovering that embedded and mobile computing researchers are also interested in memory access [Lebeck et al., 2000]. In this case the question is how many pages of memory they can slow down before application performance becomes unacceptable. The impetus is that some advanced RAM chips include

[3]The present example could be extended further once it is realised that the memory to disk connection is itself a complex system. Operating system buffering of data blocks, driver caches and their operation, hardware caches on the disk and even the mechanical operation and data distribution within the drive will affect performance. However since this section focuses on CPU to memory delays they can all be subsumed under the description given.

power saving modes which directly trade access latency for power saving, and for a device running from a limited power source lower energy consumption means the system can function for longer periods. In many ways the calculation is similar to that made for virtual memory in that the page can be moved to a lower class of memory without cost to the extent that it is infrequently accessed. If the heuristic fails then it will take a substantial number of processor cycles to bring the memory back up to active status.

Another widespread reason for non-uniform memory access is the growth of multiprocessor systems. As such systems grow, any shared resources (the bus was mentioned earlier as an example) become potential bottlenecks to scalability. This concern is mitigated by making the connection between the components indirect. A uni-processor system will almost certainly have a single bus to a single memory. A large SMP system will have many processors each of which wants multiple paths (buses) to multiple memories. This separation of elements allows for more cumulative bandwidth over the system as a whole.

This separation continues into the realm of having multiple physically independent but co-operating systems. The observation is that once the processors, memories and inter-connects reach the commodity level the cost of constructing a multiprocessor system from them becomes extremely competitive. From a supercomputer point of view the communication latency between nodes is abysmal. However if the algorithm being considered is extremely CPU intensive, with moderate to low requirements for inter-node data sharing, then this can be a viable alternative. This is the domain of distributed systems, which includes systems that provide the illusion of global shared memory for the programmers convenience [Raina, 1992]. It is fair to comment however that some of the more well known and commercial [Chien et al., 2003] efforts, have succeeded primarily by picking their problems very carefully. Tasks that naturally devolve into entirely independent sub-tasks are perfectly suited to the sort of cluster computing this environment can offer. The good results for these tasks need to be balanced against how many tasks fit this ideal profile and how well it operates with less ideal tasks.

It also becomes the case that as the system grows in this fashion the path from any given memory element, to any given processor, is likely to show increasing variation. This translates into different memory accesses exhibiting variation in access latency. It is also the case that reducing this variation tends to require more system infrastructure (more paths and more tightly coupled components) which translates directly into a more expensive (in dollar terms) system.

Some systems, known as non-uniform memory access (NUMA) [LaRowe et al., 1991] systems, are more willing to let this variation show. Part of the argument in support of such systems once again rests on the basis of locality. Most processors can be expected to spend much of the time working on a reasonably small amount of data. In addition, for many algorithms, this data is specific to the process rather than shared over the whole system. In such a case a moderate amount of local memory, with good access properties, can substantially conceal the disadvantage of the slower access that access to the systems other memory regions exhibit.

This advantage is only true if the application and the operating system in its role of allocator of memory is coded to take advantage of this diversity. Since many applications are programmed on machines with uniform memory access this property cannot be assumed. The problem is also complicated because any application running on a massively parallel computer is likely to be running a process that is shared over multiple processors, in other words a parallel program. This means that some data will be shared. If this data is automatically assigned to the most optimal position for one processor then it is automatically sub-optimal for all others, in addition to taking up memory that would be better used for truly local data structures. It is even possible to visualise a naive dynamic system wasting a great deal of time, and generating substantial spurious traffic, as the data chases from one memory to another trying to find a single optimal (and non existent) local memory.

9.4 Shards Modelling

When the hardware underlying the memory system is considered in detail it quickly becomes obvious that its behaviour is quite complex and dynamic. The hardware designers and operating system creators have both been trying to construct a system which has as many positive attributes and as few negative attributes as possible. Inevitably decisions have been made based on assumptions about expected usage of the system because many design decisions will not have a single optimal solution but will depend on subsystem interaction and the specific nature of the demands placed against the system. Since the hardware and operating system software are created well ahead of use these decisions are made against expected and generalised patterns.

This conflict between knowledge on the part of the programmer and the restraints of the API can be seen in the literature when code optimisation is considered. The application programmer has a lot of information about the usage of each memory block they allocate. The generality

of the API does not allow this information to be communicated to the system and limits how much the programmer can know about the subsystems and their run time state. For example, much work has been done on optimising performance constraints in CPU intensive scientific code [Kowarschik and Weiß, 2003], through an awareness of internal system behaviour. However these algorithms, unable to directly interact with the language syntax, compiler, operating system or hardware are forced to rely on manual and complex cleverness in the structure of their own code which may be invalidated or even result in negative behaviour when an element of the system changes. The application code is trying to guess how the operating system is implemented while the operating system attempts to tune itself to the code's access patterns gives a substantial possibility for each effort to confuse the other.

The operating system designer faces the same concerns from the other side of the API. The operating system receives allocation requests from a process but the simple interface gives it relatively little information on the intent of the allocation and the usage that will result. The operating system will have to use the actual allocation call as a starting point only. Instead the behaviour of the system must be derived statistically from the observed behaviour. The problem is that deriving usable information from this source is complex, computationally expensive and prone to being incorrect. The operating system could enter a situation where it contains a lot of complexity which has expended a lot of computational effort that leads to decisions which either do not improve the performance of the application by enough or even cause a negative interaction which makes it less efficient. In many cases keeping the mechanism simple, and not including too much complexity and overhead trying to perform optimisation on poor data, may ultimately prove the practical decision.

For an OS designer each of these subsystems and design trade offs become opportunities for optimisation or obstacles to be mitigated against. They are normally both concealed and inaccessible if interaction is restricted to the functionality provided by the API. However this interface can be worked around and the Shards system allows this work around to be integrated and applied selectively in a form suitable for reuse.

The operating system designer considers the information on the hardware and sees a number of possibilities for optimising performance using application context. Unlike the software systems introduced in section 9.1 which are mostly concerned with the process of allocating space the hardware will be involved in determining the speed of access for each use of memory. And while

it is fast the difference between the ideal and the worst case could easily become noticeable over the cumulative number of accesses. The designer identifies the following:

1. It is an advantage if data items align, as far as possible, with the boundaries of cache blocks, memory pages and zones. This allows a data item to be accessed with the minimum number of blocks brought into the cache or TLB.

2. It is an advantage if data that will be accessed at similar points in the program execution is contained within a single cache block so that it is loaded into the cache together. This approaches maximises the possibilities of spatial locality.

3. It is an advantage if data on multiple blocks that will be accessed at the same time occupies blocks that are on different cache lines. This will reduce the concern of a cache collision causing each block to flush the other from the cache when it is accessed.

4. It is an advantage if data that will be exclusively or rarely accessed occupies the same cache line. This means they will cause a cache collision but this is acceptable because the designer knows that this suits the access pattern. Arranging for this collision to occur means the block being flushed from the cache is probably of a lower value than most of the content in the rest of the cache.

5. It is an advantage if data that will be accessed on a multiprocessor system does not share the block with data that may be active on another processor. This would cause false sharing which can be expensive on a multiprocessor machine. This concern becomes focused if the data items are shared memory locks.

6. It is an advantage to keep data that will be active at the same time and for long periods of time on the minimum number of pages and aligned to their boundaries. This reduces contention and the risk of eviction from the cache, TLB cache or the page being swapped out in a low memory situation.

7. The architectural design of the hardware may mean that not all memory pages are created equal. Some of them may be cheaper slower memory (a NUMA system), or the pages may be distributed onto another physical system, they may be slowed to conserve energy or may have a narrow access pipeline shared between multiple CPUs causing interconnect contention. All of these will cause different pages to have different access speeds.

8. Any resource that uses a cache model of optimisation allows the possibility of performing an action that will bring required data into the cache ahead of access. This gives the possibility of the access suffering no performance penalty as the content it needs has been pre-fetched. This includes the cache, the TLB cache, memory pages that have been swapped out or memory banks that have been idled to save power.

The designer realises that the current analysis approach is too simplistic. The majority of the items are above are dependent on the order and timing in which the data items are accessed. Translating the operations as they occur in the source file ignores execution flow which can be expected to cause the actual flow of operations to look very different. The designer may be able to make some deductions from the arrangement in the source file but this is a substantial amount of effort and inexact. The application author could place hints in the code, indicating which elements work as blocks, but this is a substantial amount of effort. The best solution would be an automated analysis of the program flow.

There are both commercial products and research papers on doing code analysis and it is a complex field. The two approaches are static analysis [Ding and Kennedy, 1999; Chilimbi et al., 1999; Calder et al., 1998] in which the program manually traces the expected execution of the program and dynamic analysis [Uhlig and Mudge, 1997] in which the code is executed and its operation observed. Dynamic analysis is often enhanced by *instrumenting* the code so that it contains additional code to generate information about its state as it runs, although this does change the performance characteristics of the code being observed.

The Shards system offers few advantages for performing static analysis. It could in theory be done by parsing the program being considered into the data chain and then constructing filters that follow execution. However the Shards data chain is designed to be general, based on linear progression and transformation of source code. For a process as complex as static analysis a custom data structure and direct access rather than using filters is likely to be much more effective. The Shards system is better suited to supporting a filter that executes the analysis software and integrates the output.

The Shards system is more useful for assisting in dynamic analysis. Filters can be created that inject instrumentation code into the provided source, compile the code, trigger execution and integrate the output, though even here it is still convenient to have some externally provided code linked in to do the analysis as the program executes and summarise the results.

The end result of the analysis process, whichever approach is taken, will be a trace of the operations or a profile. A trace gives the sequence of operations while the profile gives the number of occurrences for each operation. The trace is somewhat restricted in that it can grow very large whereas the profile is bounded by the size of the program. The profile is useful for detecting hot spots in the code that consume a large amount of processing time which may indicate an opportunity for optimisation. This optimisation would occur at the source code level, since it will require the application programmer to reconsider their program structure, and is unlikely to be automated. The trace, and the large volume of data it provides, has more potential for automated analysis. In this case it is possible to determine for each dynamically allocated memory block which other blocks are frequently (against the total number of times the block is accessed) accessed within some given number of memory accesses. For example, if memory blocks X, Y, and Z are examined for occurrences where each access of the block is followed by an access to one of the other blocks in close proximity, filters can be created to integrate that information into the data chain.

The result is represented, in an extremely simplified form, in listing 9.2. It must be remembered that there are large volumes of other information not related to dynamic memory access that has been omitted. The data chain shown here also does not represent the trace, it is a translation of the source code file, but an attempt has been made to show that X and Y are often accessed together whereas Z is independent. The results of the trace analysis have been integrated into the data chain in the form of data items from the family memory_cache. These additional data items, added at the point the reference variable is declared, show a shared locality in that the two items are often accessed together over their lifetimes.

With the sample information in listing 9.2 it is possible to demonstrate an example of how optimisation will be performed. The filters have been expressed as pseudo code, as introduced in Chapter 7, in order to keep the listing relatively brief. Even this small sample demonstrates how filters will interact. There will be multiple filters that are members of one family and complementary in operation. Filters will respond to items in the data chain (either as a search target or a test), modify the environment or data chain, and pass control over to a filter which executes the next processing stage. The data having been processed is defined by removing or modifying the data item which is the trigger for action. Searches are represented by an "if sequence" string with a manual stepping through data items being demonstrated by the

Listing 9.2: Example 2 Initial Parsing

```
...
((memory,data,ptr),X)
((memory_cache,sharing,X),Y)
...
((memory,data,ptr),Y)
((memory_cache,sharing,Y),X)
...
((memory,data,ptr),Z)
...
((memory,alloc_heap,),X,N)
((memory,alloc_heap,),Y,N)
...
((memory,reference,named),X)
((memory,reference,named),Y)
((memory,alloc_heap,),Z,N)
((memory,reference,named),Z)
((memory,free_heap,),Z)
((memory,reference,named),Y)
((memory,reference,named),X)
((memory,free_heap,),X)
((memory,free_heap,),Y)
...
```

memory_cache_share filter. In the pseudo-code Vn (e.g. V1) is a variable.

One simplifying assumption in this example is that the analysis program does not identify a sharing relationship where the interaction can not be fully followed. For example if the declaration or allocation is in a conditional, loop or a sequence of function calls that the analysis program cannot fully follow it will do nothing. Thus the example above can be considered a linear sequence of code with the more complex program flow either occurring outside of the code shown or not interfering with the given sequence. A more advanced analysis program, and more complex filters able to use the information generated, may be able to do more but this quickly becomes highly complex.

Listing 9.3: Memory Cache Filters

```
// find a pointer identified by the traces as being related to another variable and make it
// the root of a memory block (an aligned memory page).
filter memory_cache {

    if sequence ((memory,data,ptr),V1),((memory_cache,sharing,)) is found

        // consume trigger part
        dc_next()
        consume ((memory_cache,sharing,),)

        // reserve a larger block
        if (environment:NUMA == TRUE)
            // instead of malloc use a numa call
            write ((process,function_call,start))
            write ((memory,reference,named),getpagesize())
            write ((literal,int,),environment:NUMA_FASTEST_ZONE)
            write ((process,function_call,call),numa_alloc_onnode)
            write ((memory,data,ptr_assign),V1,internal_function_return)

        if ()
            // otherwise use valloc
            write ((process,function_call,start))
            write ((memory,reference,named),getpagesize())
            write ((process,function_call,call),valloc)
            write ((memory,data,ptr_assign),V1,internal_function_return)


        // store the base address and allocated size in environment
        modify(memory_cache,size,V1),0)
```

```
        // clean up references
        schedule_filter (memory_cache_clean,V1)
        schedule_filter (memory_cache_share,V1,internal_function_return)

        // may be more variable blocks, so this is a loop
        schedule_filter (memory_cache)

    else
        end filter;
}
```

The initial filter has the base name for the filter family (memory_cache) and also drives the process. In larger filter families it would probably focus only on scheduling the correct sequence of sub-filters. It searches the application load looking for segments in the material in which a pointer reference is followed directly by a data chain item indicating that there is at least one sharing relationship. This relationship having been drawn from the results of the dynamic analysis process and placed into the data chain. Having found a matching sequence it removes the sequence, which also avoids it being used as a trigger in a later pass through the source code, and adds some operations to allocate a larger block of memory. Immediately allocating the memory following the declaration helps to resolve issues in which declaration order does not match allocation order.

Listing 9.4: Memory Cache Filters

```
// need to remove all sharing items targeted against the root since it came
// first and is now a host for sharing rather than a participant
filter memory_cache_clean(V1){

    if sequence ((memory_cache,sharing,),V2) and V1 == V2 is found

        consume ((memory_cache,sharing,),)

        // loop
        schedule_filter(memory_cache_clean,V1)

    if ()

        end filter;
}
```

Since the first declared variable with a sharing relationship has been defined as the foundation for a grouped variable block it is no longer available to share space with another variable. In other words the analysis program will identify a transitive sharing relationship since multiple variables occur together frequently and are equal in status. Having selected one element of the shared relationship to be the base element, the relationship is now directed, with the goal being for other elements to share the space that has been allocated. This filter iterates through the chain and removes sharing relationships that reference the variable we have selected.

Listing 9.5: Memory Cache Filters

```
// schedule filters for all values shared with root
// arguments are root identity and base pointer.
filter memory_cache_share(V1,V2){

    // setup assignment for the base assignment
    schedule_filter(memory_cache_assign,V1,V2)

    // assign shared values that can use this root pointer
    if (memory_cache,sharing,V3),V4) and V4 == V1
        consume ((memory_cache,,))
        schedule_filter (memory_cache_assign,V1,V3,V2)
        dc_next();
    else
        end filter
}
```

This filter steps through the data chain looking for dynamic memory variables that have a sharing relationship with the selected "root" variable. This includes the root variable itself as can be seen from the first line. It then looks for the declaration of variables that have a sharing relationship with the root variable. On finding one it consumes the data indicating the sharing relationship since that information is being used to trigger changes. This both stops the trigger being recognised again and reflects that the goal is to modify, and thus invalidate, the relationship represented by the sharing data item. The change itself is implemented by scheduling a filter using the name of the variable that may be able to share a memory space with the root variable.

Listing 9.6: Memory Cache Filters

```
// correct related variables and their references. Arguments are
// value being sought and base pointer
filter memory_cache_assign (Vroot,V1,V2){

    Vcount = 0;

    // found the allocation which can become an assignment
    if sequence ((memory,alloc_heap,),V3,V4) and V1 == V3 is found

        // get the current filled size
        V5 = access(memory_cache,size)

        // abort if we've filled a page
        if (V4 + V5 > getpagesize())
            end filter

        // if not it is now an assignment
        consume ((memory,alloc_heap,),,)

        // and no longer a potential sharing host
        schedule_filter (memory_cache_clean,V3)

        // write the new operation
        write ((memory,data,ptr_assign),V1,V5)

        // add the size of this item to the filled portion
        // of the memory block
        V5 = V5 + V4
        modify(memory_cache,size,V5)

    if sequence ((memory,free_heap,),V2) and V1 == V2 is found

        // write
        consume ((memory,free_heap,),)
        modify (memory_cache,count,Vcount++)
        write ((memory,free_heap,),Vroot)

        // shrink alloc size
        V5 = access(memory_cache,size);
        V5 = V5 - V4
        modify(memory_cache,size,V5);

    if ()
        schedule_filter(memory_cache_fixfree,Vroot)
```

```
        end filter
}
```

This filter performs the process of modifying the allocation operation to use the newly created shared memory block. The new process will effectively replace an allocation operation with claiming a small section of the memory block to take its place. As mentioned this code assumes there is a direct relationship between allocation, use and deallocation with any case where this is not certain being rejected by the analysis portion as a potential sharing relationship.

The second part of the search finds matching free operations. These are a trigger to release the amount of memory claimed. This is acceptable because we are handling all of the allocations and deallocations for one variable name at a time so there will not be interaction between different variables claiming memory. This section of code also rewrites the free operation so that all deallocations are relative to the main block. This is to resolve the confusion over which free operation is the final one that can remove the entire block. A count of the free operations encountered, which includes the base variable deallocation and the occurrences of the variable we are modifying, are kept and the final filter called to clear up duplicates.

Listing 9.7: Memory Cache Filters

```
// delete the earliest free
filter memory_cache_fixfree(V1){

    Vcount = access(memory_cach,count);

    if sequence ((memory,free_heap,),V2) and V1 == V2 and Vcount > 1 is found

        Vcount --
        consume ((memory,free_heap,),)

    if ()

        end filter
}
```

As mentioned above this filter looks for all free operations against the root variable. Duplicates have been created in the previous step as it is dangerous to deallocate the block when it may still be in use by one of the hosted variables. The solution is to deallocate the block when the last use occurs in the code. This function will be called once for the base variable and once

Listing 9.8: Processed Data Chain

```
((memory,data,ptr),X)
((process,function_call,start))
((memory,reference,named),getpagesize())
((process,function_call,call),valloc)
((memory,data,ptr_assign),X,internal_function_return)
...
((memory,data,ptr),Y)
...
((memory,data,ptr),Z)
...
((memory,data,ptr_assign),Y,X + X_size)
...
((memory,reference,named),X)
((memory,reference,named),Y)
((memory,alloc_heap,),Z,N)
((memory,reference,named),Z)
((memory,free_heap,),Z)
((memory,reference,named),Y)
((memory,reference,named),X)
((memory,free_heap,),X)
```

for each variable that is being hosted by the allocated block. The final result will be only the last free being left in the code.

In optimisation terms the filters use the sharing information placed in the data chain to make a simple optimisation. Specifically when they find a dynamic memory variable that has an unprocessed sharing identifier that value becomes a root node. The filters know they are optimising for speed over size or they would not have been scheduled. The root node allocates an amount of memory determined by the design focus of the project, in this case a page of memory, and occupies the start of it. Other variables that have a close temporal relationship in the analysis trace have their own allocation functions replaced with a reference into a section of memory in the larger block. When the block is filled or there are no more variables with a close relationship that have not been integrated the process restarts. When there are no triggers at all the filters are complete and the next filter defined in the data command structure will operate. The end result once all the filters have run to completion is shown in listing 9.8 which can be compared with listing 9.2 to see the differences. This would then be expanded into the application source code and compiled.

9.4.1 Operating System Modelling

The optimisation example given is both trivial as a mechanism and fairly complex as an example. A large part of the complexity stems from the fact that performing static analysis on application code is a challenging task, a task that the structure of the Shards system does little to make easier. Even with the simplifying assumption that the external analysis program would reduce the challenge of the cases available the example given is still fragile. Application code could be written that would cause the simple heuristics used to generate erroneous modifications to the application load. This would probably be resolved by reducing the ambition of the optimisation, finding the edge cases, recognizing them and doing nothing. This solves the problem but also reduces the usefulness of the solution.

A better solution is to allow some of the decisions to be dynamically resolved at run-time. As an example the previous filters depended on the analysis tool not flagging as variables that had a high co-locality but which the filters could not handle. For example if the memory is being dynamically and conditionally allocated the example filters, which make no attempt to follow code flow, are unable to detect or handle this case. The analysis program could be extended so that it provides additional information, and the filters made much more complex to include it. For many problems the even simpler solution is to use a system call and allow the operating system to assist in implementing the solution as it has a superior understanding of the dynamic state of the process.

The question then becomes what can be most usefully determined through designer intervention or application code analysis and what is better suited to a solution within the operating system. An optimisation that can be identified and resolved entirely at the language level gains in performance due to no additional run-time cost at the cost of substantial complexity in the analysis and implementation phases. This benefit can often be fairly small in practice and often resolved as easily by rewriting the code to directly include the chosen optimisation.

The best potential for substantial benefits are where the analysis process can identify that the code interacts with the design goals of the system. Unlike a straight performance optimisation this may not be something that concerned the author of the code. Linking this interaction with an operating system facility that expresses the systems focus on this type of interaction has the potential for efficiency gains without excessive analysis.

To continue with the example presented in the previous section the optimisation is grouping

various memory elements so that they are treated as a single block. It implements the solution using standard operating system calls to provide a block of memory. It then is forced to confront the complexity of correctly managing this memory to reflect the dynamic execution of the process. Meanwhile the operating system has no involvement. A block of memory was requested for purposes the operating system has no way of divining or interacting with. Since only standard operations are being used there is little opportunity to tailor the operating system to operations of interest.

One way of influencing the operating system would be through the process of tuning its construction. For example the sharing relationships could be analysed by the filters to determine how many variables there are, their average size, lifetimes and of course locality. The output of the filters might be some tuning variables in the Makefile of the operating system to incorporate these details in how it handles memory, such as determining what internal mechanism it should use to satisfy memory allocations.

The problem still exists that this is statistical and imprecise. The operating system is still limited in how much information it can divine about any specific usage. An allocation may be for a variable which has a potentially optimisable sharing relationship or it may not. Without this information it is forced to treat any individual call as a generic operation. This is resolved if the action of the filter is to differentiate the system call either through renaming it or adding additional arguments. This allows the filters to focus on analysis and directly communicate this with the operating system. The operating system in turn allows implementation to be simplified, to directly access system state and to interact outside of the limits of a single process. This is referred to as a *hook* in that the solution of the analysis phase is to identify opportunities for project design goal specific optimisation. The decoupling of identification and solution allows for much simpler analysis, better handling of dynamic conditions and most importantly alternative solutions to be trialled and selected on the basis of the projects design priorities.

In the example given this could be demonstrated by assuming the operating system has been extended to include the *shared_block()* operation. This command accepts an integer identifier used to gather variables with a high co-locality (they will all have the same integer argument) in addition to a size argument for the amount of memory needed. The operating system recognises when a free operation references a variable which might be within a shared block, uses reference counting to know when the entire block is empty and can be freed, and can deal with allocation

requests when the block is full and alternative allocation must be used.

With this addition to the operating system much of the complexity in the example vanishes. Dynamic paths of execution in the code immediately become a non issue as the solution itself is dynamic. Concerns for which of the variables with a sharing relationship comes first and which free comes last are gone. Concerns about what size or type of memory to allocate can be left to the operating system which has a superior knowledge on the allocation possibilities available within the system. The filters need only group sharing relationships together and modify the allocation call to use the new system operation. This will still require the analysis software to determine co-locality but neither it nor the filters need to consider program flow greatly simplifying the implementation. If the operating system has a novel implementation solution this can be applied without invalidating the existing analysis stages which encourages reuse of both parts.

The new filters would produce the output shown in listing 9.9, which can be compared with listing 9.8. The solution is much cleaner in that the solution needs to make fewer changes and assumptions to the code while being able to cover a much wider range of run-time conditions. There is no allocation of memory, no duplication in handling, instead the focus is entirely on identifying that the variables X and Y have a sharing relationship which is a potential for optimisation. The implementation is either up to a later filter, if the decision is made to solve it within the process image (as per the old process) or a simple filter that translates the shared_block data item into a matching call on the newly provided operating system functionality. This could also involve modifying the build process so that the functionality needed is linked into the operating system.

9.4.2 Operating system opportunities

The Shards system is now able to integrate an expression of the project's design goals, analysis of the content and operation of the application load and the selection and utilisation of operating system mechanisms. This gives a great deal of focused design flexibility. A solution can be considered in terms of how it matches the goals of the project and at which level of the system it can best be integrated.

This includes the degree to which the complexity of the solution is worth the effort of implementation. A solution that involves an easy translation of a operation within the source-code

Listing 9.9: Example 2 Initial Parsing

```
...
((memory,data,ptr),X)
...
((memory,data,ptr),Y)
...
((memory,data,ptr),Z)
...
((memory,shared_block,1),X,N)
((memory,shared_block,1),Y,N)
...
((memory,reference,named),X)
((memory,reference,named),Y)
((memory,alloc_heap,),Z,N)
((memory,reference,named),Z)
((memory,free_heap,),Z)
((memory,reference,named),Y)
((memory,reference,named),X)
((memory,free_heap,),X)
((memory,free_heap,),Y)
...
```

requires little analysis. More dynamic operations require more complex analysis but that analysis can be simplified by using programmer provided hints, allowing the operating system to make some decisions at run-time or even deciding to reduce the scope of the optimisation. Analysis of the code may also determine how valuable, or detrimental, the effects of the planned optimisation may be. If the needed filters already exist and are suitable for reuse then the possibility of experimentally determining their value with negligible implementation effort becomes a possibility

To demonstrate the design possibilities that this can offer it is worth considering another example. This example will also be within the domain of dynamic memory management to avoid introducing additional technical context. It will also build on the approach presented in the previous example but focus more on how code within the application load can be productively linked with the underlying mechanisms of the operating system being developed.

The specific example will come from the Shards implementation itself. This is not to claim it is superior or special code but it does make substantial use of dynamic memory. The particular source file is atom.c which handles mapping a string to a unique integer and vice versa. It stores

Listing 9.10: Code Sample

```
struct cnode_t;

typedef struct cnode_t {
    char* section; // a section of a word.
    struct cnode_t* next; // the next section.
    struct cnode_t* prev; // the previous section (for index -> word)
    struct cnode_t* down; // the next alphabetic section
    struct cnode_t* across; // links word terminations by count order.
    int count; // the number given to this word.
} cnode;
```

string segments so that common subsections and prefixes are not duplicated. The central data structure being used is described in listing 9.10.

The software dynamically allocates a new structure when a new string subsection must be added to the list. Its actual usage in the code is not relevant to the discussion and will not be covered. The interesting part from an operating system optimisation point of view is that there will be many of these structures created in the lifetime of the program, they are generally permanently allocated for the lifetime of the program (it is safer to keep the mapping even if the software using it believes no one else will access it), they are of a constant size since the string subsection is stored separately and they are smaller than a single L1 cache block (24 bytes on a 32 bit system). The creation of these data structures will be separated by a variable length but probably small allocation to hold the string segment. Since they are being created dynamically as needed this means they may also be separated by any allocations between new cnodes being created. The result of this is they will be spread irregularly throughout memory.

The coverage of the process's handling of memory in Section 9.1 indicates these small allocations will go into the heap at which point their handling is outside of influence by the Shards system without reconstructing the compiler. It is also likely the compiler's internal memory management will not attempt to infer usage information from the allocation and will simply attempt to pack them for access efficiency and minimal fragmentation. If the Shards system can capture this usage information then it is possible to engineer a solution that may be more efficient in terms of how these memory allocations interact with the cache hardware.

The specific opportunity being that these allocations of memory are related and should be densely packed. The optimisation heuristic of *cache colouring* which seeks to evenly distribute

data in the cache to avoid collision works directly against this. It is designed to avoid collisions so that data remains in the cache as long as possible on the basis that it may be access again. However on examining the code it is likely the operation will be doing a traversal and lookup which is likely to mean the useful lifetime of an individual node is very short, most likely a single access. The costs of cache colouring present no benefits under the most likely usage pattern. This is an expected possibility of a heuristic based solution to optimisation which cannot guarantee to be optimal, or even not sub-optimal, in all cases. The technique of *cache alignment* in which data items match cache line boundaries is less clear. It may be advantageous, in allowing optimal access to a data item, but this must be balanced against a reduction in the density of storage. If the operation in question is a traversal then having a section of the next data item in memory is an advantage. The ideal balance becomes a design decision. An example of both of these memory techniques in practice can also be found in the implementation of the Linux Slab allocator [Bonwick, 1994].

In the previous example the analysis phase was performed using dynamic profiling of the application load. For this example the solution could use the simpler process of looking through the code for any malloc operation using sizeof(cnode_t) as the argument to malloc. This would provide enough information to identify the hooks for a modification without complex dynamic analysis. Alternatively, if there were ambiguity such as multiple structures using the cnode_t data type, the analysis could rely on programmer hints to guide a solution. This approach trades off exact control and specific application of the modification with the additional effort of modifying the application code.

In this example all members of cnode_t will be elements of a single data structure since the program only has a single lookup table. However some allocations are just for temporary storage and it is also desirable that the modification of handling does not occur for all malloc operations within the project. So there must be some way for the programmer to indicate which data items indicate triggers for customised handling. Once again this could be determined through dynamic analysis of the lifetime of the data item and the number allocated but this is complex. Instead we request that the programmer uses a #typedef to create an alias to the data type which gains from customised handling. If compiled without Shards there is no difference, both the typedef and the base type will generate the same size. However Shards can respond to the typedef only and ignore malloc operations that do not use the correct format in the typedef name or are base

Listing 9.11: Hint based optimisation triggers

```
// memory allocation that can gain from optimisation
typedef struct cnode_t shards_memcache_cnode;
struct cnode_t* data;

// malloc operation that will be transformed.
data = (struct cnode_t*)malloc(sizeof(shards_memcache_cnode));

// malloc operations that will not be transformed.
data = (struct cnode_t*)malloc(sizeof(struct cnode_t));
data = (struct cnode_t*)malloc(24) // size of struct cnode_t
```

type allocations. In this case the string shards_memcache_ will be used as a prefix recognised by the filter.

An example of this approach can be seen in listing 9.11. In this sample of code there are three malloc operations that from the point of view of the compiler are identical. And in the absence of Shards processing will operate as expected. There is not even a performance cost as the typedef will be resolved and removed by the compiler. However the Shards system can use this construct as a trigger for optimisation. A filter reacting to the structure and presence of the typedef in the first malloc operation can be certain the call is suitable for optimisation. This is because the code, through this modification, contains hints that resolve ambiguity. The same could also have been done with a structured comment that identified struct cnode_t as being suitable.

The filters could implement the optimisation at the language level, which is to request from the system a large block of memory and then manually pack the individual memory requests within this block. This would allow the related data items to be contiguously stored in memory. However there are several arguments that encourage this solution to be solved at the operating system level.

The first is that doing so would enable the operating system to align the allocation block to a page boundary. Since the data items will be referenced together efficient allocation with reference to the page table structures, and more exotic options such as pinning or pre-loading the page table, have very similar benefits to those of the cache. Page tables are generally not available at the application level. In addition the calls that would allocate memory blocks of that size will reserve all the memory up front. The operating system is able to reserve a page table

but not allocate physical memory to it until required. This allows a large region of memory to be potentially available without excessive waste if the amount needed proves to be much smaller.

The second is that, as in the previous example, the memory allocations are dynamic. Detecting malloc operations that exist within loops or conditionals requires a great deal of program analysis that is complex and can also be fragile in the face of unusual structures in the source code. Since the main optimisation interest is in terms of access, rather than allocation, it is much easier to translate the malloc calls into calls on functionality that will be included at the operating system level.

The optimisation is thus implemented in two parts. One filter initialises the system by placing an operating system call to allocate a large block of potential memory. This will be identified by a memory address so that multiple variables can be the subject of the same allocation process. Each malloc operation that contains the hint is then converted into a function call which references this address. At runtime the call physically allocates a block of memory and places it into the pre-allocated page table. With the operating system's knowledge of the cache structure, and the design goals, it may choose to waste a small amount of space to keep data items cache aligned, but the over-all goal is likely to focus on dense and contiguous packing both at the page and cache level.

This approach, in which the code identifies a need and Shards connects it with a solution implementation encourages creativity in considering potential solutions. The example above can be combined with code analysis to make it more sophisticated in how it responds to the structure of the code. As an example the programmer knows the data structure will most often iterate through the next pointer when doing a search. The other pointers are more random in their access patterns. The filter could potentially be extended to recognise allocations that will be assigned to the next pointer and utilise this information. This could be by the programmer explictly specifying that the next pointer indicates the most likely path. It could be by the system designer giving the filter the variable name of the next pointer. The filter could even try automated heuristic approaches like guessing which variable name is likely to be associated with the most likely path or looking for pointer access in tight loops indicative of searching a linked structure. Whichever mechanism is used the operating system component can then organise the block so items that are connected by a linked pointer are arranged consecutively. The implementation can be constructed through using different regions of the reserved page

table for the two types of allocation request.

Another operating system optimisation possibility is cache or page pre-fetching. Since the allocations have been structured to be contiguous it is likely that a traversal may also want the next page or cache line in order. This probability becomes even higher if the data has been organised as above since that means the common access path for traversal operations is even more densely packed. The hardware provides an operation to make use of this knowledge so that data likely to be required can begin the process of being readied in the cache before it is needed.

> Software pre-fetching requires a programmer to use PREFETCH hint instructions and anticipate some suitable timing and location of cache misses. ([Intel64:2014], Section 3.7.1, p.147).

If the system knows that pointers often searched in sequential order are being accessed it might be worth pre-loading the next memory block worth of pointers in advance. This can negate the performance penalty of both a cache miss and the transfer of the needed memory block if the prediction is accurate and a pre-fetch command has been given. In practice the effort of using the full potential of cache pre-fetching runs into very similar concerns as those faced by the Shards system. Adding the argument manually requires a substantial amount of programmer analysis and effort, adding the command automatically requires advanced code analysis tools and both require a customised compiler. This use of a custom command and compiler will also limit the hardware platforms on which the modified application can be executed.

The example being discussed provides a demonstration of how the Shards system can make integration of new idea for optimisation much easier. Since the optimisation process knows that a group of data objects are frequently accessed sequentially and both densely and contiguously packed much of the information required to make use of the possibility of a cache (or even memory page) pre-fetch operation is already available. An additional filter that identifies every access on a managed data item and includes a pre-fetch operation for the next block in memory has a high probability of a beneficial outcome. Complex program flow analysis is not required as the access and pre-fetch operations will always be co-located. If conditional code flow modifies whether the access occurs then this will also skip the pre-fetch operation. The calculation of memory locations, which may become complex if the address is being manipulated, can be

simplified by making the calculation a dynamic calculation at run-time, using a simple heuristic and accepting some incorrect pre-fetch operations or not placing a pre-fetch operation when the situation is ambiguous. The Shards system also separates the analysis from the implementation allowing a final filter which converts the abstract operation into a compiler specific variation or removes it if the required operation is not available. This allows the solution to be more widely applicable and automated.

The facility being discussed could be evolved further as new approaches or environments are integrated. For example should the memory optimisation process work differently if the program will be running in a multiprocessor device? It seems likely that having memory locks and multiple processors seeking to access the data items would be interesting territory for new idea on how to integrate that sharing given the information available. This ties into concepts like lockless transaction [Shelton, 2011] which provides additional interesting possibilities. In that context the idea of memory zones, being able to split data items into more or less valuable blocks and using this information to optimally distribute the data over a NUMA system, raises yet more possibilities. Using the information and implementation in an existing Shards filter as the foundation for implementing new environments or possibilities, and thus producing either more or superior filter possibilities, is the positive feedback cycle that Shards seeks to provide the basis for. Even if Shards itself proves to be mechanically inadequate encouraging the idea of a unified platform for this process, and providing a foundation for its own replacement, is valuable.

9.5 Conclusion

This section concludes the demonstration of the Shards system. It is intended to be in support of the previous chapters that introduced the Shards mechanisms and how the system environment is captured and manipulated. It is not intended to offer any particular new insights into memory management. It may be taken as a demonstration that when an operating domain is examined closely the complexity, and thus opportunities for manipulation and optimisation, can grow quickly.

Chapter 10

Conclusion

The Shards process provides a structured approach to system design. A prospective system designer does not need to start from scratch. Instead they can use the Shards process to start capturing the context and goals of the system which will determine if a development effort is justified. This information can be understood by filter families which provide packaged but efficient functionality available for reuse. Re-use allows the project to focus on what is novel and unique, and get more quickly to the point where a prototype system can be tested so that project risk and uncertainty are reduced.

The system design effort produces new filter families, new data that can describe a system and be used by filters to optimise the functionality they represent, more sophisticated automation within a filter and new implementations to extend existing filters. New insights are produced at the design and architectural levels through being able to explore the interactions between custom system design goals, the systems constructed using filters and their measured outcome in the target environment.

A researcher in the field of system design can productively refine any part of this process without the need to construct an entire system. A filter can be studied in isolation, or as part of a limited sequence, and improvements investigated and implemented. Small research efforts become more practical through reducing their scope and are more valuable in that the product can be used in future development efforts for full systems. The design experience and system insight the developers gain in the process will also be reflected in the components they can reuse in future projects. It is also easier to perform research at the architectural level. The modular approach allows sections of an existing Shards generated system to be more easily modified or

replaced and the new system variants tested.

The result does not seek to replace the existing general systems. General systems focus on providing a single long lived platform on which software can be built. Their maturity, stability and rich software ecosystem are a large part of their advantage which there is little benefit in challenging. At the same time, the idea of considering system construction to be part of solving a specific problem and producing a customised system ideally suited to a specific environment means that the strength of the existing systems does not indicate there are no possibilities remaining to be explored.

The Shards system provides a workable framework through which these goals can be reached. Shards allows non-traditional inputs and a wide variety of functionality to be integrated into a single approach. Shards addresses the conflict between the convenience and the inefficiency of a modular approach. Shards provides a model of automation so that the system designer can focus on what is novel without losing the potential for complete flexibility in what is constructed. There is no doubt the process itself has much room for extension and improvement, even complete replacement, but it has demonstrated a new and structured way of discussing, designing and constructing useful system software. Being a structured system it has more potential to experience positive feedback cycles in which use refines and enriches the tool. The ability to build upon this foundation is something the traditional ad hoc model of system construction could never offer.

The following sections will continue to explore these issues, some discoveries made in the process of development and discuss the immense scope for future work.

10.1 Risk Reduction

A major element in considering whether to construct a custom system would be a calculation of effort and risk. The more accurately effort is estimated and the smaller that value is, the more likely construction would be attempted. There is an additional element of risk that unexpected complexity will be discovered once the project has been initiated and resources invested. This acts as a potential multiplier on the effort involved and can even lead to the effort being abandoned. A predictable system development process in which risk can be minimised allows greater confidence that the project is worth pursuing.

The solution is to take an approach common in general software engineering, the use and

reuse of modular code, and apply it to the system development process. This reduces the need to write custom code from scratch, allows early testing and iterative development which makes the development process simpler and more predictable. The novelty of the Shards system comes in providing a unified framework which enables the possibility of abstracting a system mechanism from the system in which it originated. The conversion into a modular form must be done without loss of flexibility, and the modules use in future systems without loss of efficiency.

A mature Shards system provides a designer access to a toolbox of filter families, all of which use the same structure so as to be mechanically interchangeable. These filters can be used where the requirements of the new system are not specific as well as where the filter suits or can be easily adapted to suit the design goals. This reduces the area of uncertainty, allows optimisation efforts to be focused and can even provide filter primitives which may aid in the construction of a new filter family[1]. This also allows a minimal system to be more quickly bootstrapped so that development assumptions can be tested early which would further reduce the risk factors of the construction effort.

The advantages of modules as a foundation for building large and complex systems is not a novel observation. They are a fundamental toolkit in most of computer science. The restraint on their uses is the issue of complexity. A large collection of modules, designed to be part of an integrated system, must contain structure, consistent expression and expectations of how they will be used. The prospective user must first understand the breadth of modules to be aware of the available functionality. They must then understand the structure of the modules in order to be able to apply them. Learning all the possibilities and best application of a large software library requires significant study and experience.

Collections of code intended for reuse, such as libraries, tend to respond by simplification of the interface. Complex behaviours and internal states are discouraged. A relatively generic call is favoured over many variants, all with different behaviours and complex arguments which further configure its operation. This reduces the complexity on the expectation that the loss in flexibility, configurability and optimisation is not critical to the reuse of the functionality encapsulated within the library or module. This approach is not viable for Shards where all of these aspects are important in providing value to the process of system construction.

[1] In practice it is quite likely bespoke code would come first and be integrated with the components generated by Shards. This could then be adapted into a Shards filter once it had been proven. There are advantages to doing initial development outside the Shards framework.

The Shards system offers a solution to the conflict between interface clarity and complete control. Instead of the interface being considered as a static and generic construct, it becomes a dynamic range of possibilities. The Shards process allows for multiple variant mechanisms, each with their own control options to be considered as a single filter family. The Shards process is designed to allow the automated selection, configuration and integration of a specific mechanism and interface so that they best fit the needs of the system and the capabilities of the module. This process can respond to both the designers description of the system as well as other filter families being used in construction. As much as possible, it operates without the direct intervention of the designer, thereby reducing the complexity that must be considered before optimal use of the filter family is possible.

In an ideal system, the designer would not be required to understand the internal workings of the filter family at all. The designer's role would be to select the filter family by including its name and any desired arguments in the project file. The filter should then integrate the functionality it represents intelligently into the system being constructed. The more sophisticated the filter the more complexity it can contain and automate without relying on manual integration by the designer. The filter may perform better from the designer knowing there is information it can use and providing this directly as an environment variable or argument. The same method can be used if the designer wants to override the the automatic behaviour. Filters that cannot fully automate the process of integration and optimisation are still useful for the amount of complexity they can encapsulate. The manual intervention required to complete the process will suggest improvements to the filter that when implemented provide iterative evolution on the capabilities of the filter family.

10.2 Optimisation Reward

The reduction in the risk of system construction combines with any advance in the value the generated system can provide to make the creation of new optimised systems more viable. The Shards process works on the foundation that the interaction between the design goals of the system and the context in which it will operate must be captured as part of the system construction process.

This observation has direct parallels in the actions of an expert designer working without the Shards process. They will consider the mechanisms they know, the tools they have and the

goal of the project being considered. They will do their best to make sure that each element of the system is constructed so as to incorporate this information and contribute to an optimised system. The difficulty with this approach is scaling and scope. A system constructed in a manual and ad hoc process will have to put a lot of focus on efficiency of constructing the system in order to control project risk. A lot of time will be spent building subsystems because they are required even if they are not central to the design and optimisation goals of the system. Building tools to analyse the environment and gather information to drive optimisation will take time to build and additional time to integrate into the code being written. This leads to a risk minimisation strategy of building the system first and then using the time remaining to focus on what can be optimised.

The Shards process enables the possibility of automation which allows optimisation to become integrated into development. The designer can specify Shards filter families for the functionality that best matches the design goal. The designer can provide information about the system and the design goal and expect filter families to do some work in adapting their operation to work with that goal. The designer can also make use of the Shards framework to integrate analysis such as application load analysis to provide additional information. The more that Shards can contribute in analysis and generation of components, the more complete the foundation on which system construction can start. This allows more time to be focused on components vital to the project goal which Shards cannot provide or sufficiently optimise. The results being generated provide the identification of an area in which Shards coverage can be extended and the material to integrate into a new filter family or implementation variation.

The Shards process provides a framework into which system analysis tools and system architecture models can be integrated. This allows the system being constructed to be connected with the environment in which it will run. General operating systems are also the definition of the platform they provide. They are built around expected use patterns and heuristic estimation. A customised operating system is given value by how well it can integrated with the rest of the system and the goal for which the system has been constructed. The results of system analysis, the architecture selected, the insight of the designer and all components that will be part of the final system can provide information to drive optimisation decisions. Any system information that can be specified, defined, discovered or determined can be integrated into the Shards process. This creates a positive feedback cycle where analysis tools are given value through Shards

integrating their output while system construction using Shards provides more opportunities for analysis. The process of analysis and integration also suggests opportunities for new systems and a means of estimating their potential advantages which motivates new system construction efforts.

The other advantage of the Shards system is that optimisation occurs primarily as an integrated part of system construction. As long as operating system solutions are designed to optimise their behaviour at run-time, a virtually universal property in the field, they will suffer a performance penalty for any intelligence they contain. Since a customised operating system is likely to have efficiency as a primary focus this is a negative interaction. The Shards system encourages extensive analysis before construction and the results to be automatically integrated into the operation of the system. This allows the system to contain custom and intelligent behaviours with minimal run-time effort.

The benefit of having superior information as input to the system cannot be over-valued. Statistical analysis of memory allocation behaviour [Wilson et al., 1995], which works to judge the relative efficiency of memory allocation implementations, is an example of clever analysis from limited data, since only the final part of the execution sequence is known. The study, as with many others in the field, relies on heuristics and can only test against a representative sample. This is a weakness even in the general domain because the connection between application task and system behaviour can only be inferred. Since analysis provides limited information about system intent or the value of different outcomes there is little to guide optimisation at the system level or encourage the development of new systems. This has been an element in system design for some time with developers using simulated or captured traces [Wilson et al., 1995] to test their systems under run time conditions. Other developers use data collected at run time [Ding and Kennedy, 1999]. This profiling carries a performance cost due to the degree of precision and the duration needed in order to gain averaged information which discourages extensive analysis during the operation of high value systems.

This approach to analysis contains an inherent assumption that application demands on systems are broadly constant and undifferentiated such that the results of analysis can be expected to be broadly applicable. Improving the average performance is the focus. The Shards approach suggests that there is unexplored potential for custom systems where these assumptions are not valid. Instead analysis should be focused on the behaviours that give value to the specific

system and solutions that can be integrated into the design and construction process. The use of the application load is an example where the applications that give the system value can be analysed directly and used to inform the design of the system. An ideal architecture for the specific purpose the system has been built around can deliver large performance improvements without run-time cost and with relatively simple but appropriate mechanisms. Shards captures and automates this process of matching analysis with system construction so that it may be time efficient, system specific and potentially reused on future system design efforts.

The growth in the scope and importance of computer systems has continued to increase. The modern mobile phone has sufficient processing power and software intelligence to perform tasks that would have been in the realm of science fiction until relatively recently. With sufficient processing power becoming a commodity item, computers are omnipresent, built into a wide variety of forms and supporting a rich and varied software ecosystem. The companies that control the dominant systems gain immense market advantage and profitability from owning the foundation of these valuable systems. Exclusive functionality, extremely user friendly interfaces, good performance on commodity hardware and rich software libraries supporting development of applications all help to protect established systems from competition.

The increasing complexity of much of this software, the degree to which it must interact with the real world through sensors and local networking, and the requirements for small form factors and low power consumption provides a great deal of pressure on the underlying operating system. The system software is concealed by user friendly interfaces and integrated deeply into the device, but the value of these systems has only increased. New operating system capacities that can provide more performance and application advantages allow an even wider range of form factors, specialised appliance variants and new functionality which can give a meaningful market advantage.

Many of the areas on which the Shards process focuses are even more important in this environment. The ability to produce or iterate on operating systems more quickly and with less project risk has value in this time sensitive market. The ability to integrate more closely with the applications that give the device value works well with sensors, automated systems, computing as appliance and a market where the value is in software and much of the hardware is a commodity product. Lastly, the operating system being defined and adapted to the project goals allows the complexity of the generated system to scale down to meet the specific requirements of the target

system without bringing functionality that is not required.

10.3 Systems Theory

An additional advantage of the Shards process is that it provides a structure for expressing systems and the discipline of system design and theory. At the moment, communication of ideas within these fields is imprecise and requires verbose documentation in addition to the construction effort. This reduces cross pollination of ideas and reuse of existing functionality between different research projects. The Shards project provides a regular structure that can encapsulate a wide variety of systems and system functionality. This allows system to be defined by the Shards components they use rather than the full detail of their construction.

Describing an operating system has traditionally been problematic. Operating systems tend to be large and contain many subsystems which can have their own complex design logic as to how they are constructed. There is rarely a single 'idea' that explains all parts of a real system in a simple way. Additionally, these systems tend to be tightly coupled (have many internal interactions and dependencies) in addition to machine dependencies and interactions with the external context in which they exist.

The current method of communication is technical writing and source code. Technical documents tend to get very large because systems are constructed from so many individual components. Highly detailed description is needed to understand the functions of each part because English is not inherently rigorous. This is multiplied by the number of parts and again by the interactions. This is why the manuals for the OS/360 system are described as a "6-foot shelf" [Brooks, 1995], as page count had ceased to be a suitable index of scale for these creations.

The need for system documentation for potential users is another cost to system creation. As such there is pressure to keep the scope controlled. This means that the focus will be on documentation that must be available for operation and utilisation of the system. More abstract elements such as the design and optimisation decisions, the discoveries made during development and how this is reflected in the final product often remain only in the memories of the designers.

The other document is the source code produced if it is available for perusal. The source code is specific and rigorous, but given that operating systems are measured in millions of lines of code they only barely qualify as readable. The reason for this is because no allowances have

been made for human limitations. The computer has a phenomenal memory and as such can build a structure from the low level description of it. A human mind cannot duplicate this feat. Not only will each page be cryptic, because the context is missing, but attempting to build that high level context from the raw source would be a substantial task. It could also be argued that it is this factor which helps explain why operating systems, whose exact documentation is frequently only within the source, are rarely reused.

The Shards system offers another possibility. If source code and the functionality can be subsumed in filter families then systems can be described in terms of those filter families. The details of how they are integrated into a system becomes automated, reproducible and fine detail would not be required to express a high level understanding of the system. Design details and innovations are expressed by new filter families or extension of existing filter families which enables intent and implementation to be described separately. Once done, the Shards project file becomes a concise description of the system that can be read, expressed and compared with other systems using the Shards process.

The Shards model also includes making the process dynamic in terms of analysis and optimisation without run-time performance penalties. This allows the context in which the system will run and the design goals that shape it to be expressed. The analysis of the application load was one example but any aspect of system design and analysis could potentially be integrated with the Shards process, allowing it to be compared, contrasted and reused in future projects.

10.3.1 The "Bootstrap" Problem

The primary concern with the Shards system is the "bootstrap" issue. This reflects the problem that Shards adds complexity through its own structure and an unproven advantage through the possibilities it enables. With the system being immature and the number of filter families being non-existent, there is little advantage to making it a core part of an expensive and time pressured system development effort. This in turn reduces the ability of the Shards system to iterate, evolve and gain the filter families which would provide a system creation advantage through reusability.

The solution is to focus initially on the idea of Shards as a model for how the discipline of system design can be structured. Since research is often focused on mechanisms, parts and comparisons rather than the generation of completed commercial systems, there is a natural

shared interest with the goals of the Shards system. The change is to think of research within the field as steps in the evolution of a framework which can integrate them into a unified whole. This changes individual research projects from being incomplete efforts created in isolation into steps starting from a framework to a more improved form. It also allows research projects to be focused on discrete units of construction rather than the unmanageably large project that generating an entire system can represent. This attitude and approach to systems research is more important than the actual implementation mechanisms provided by the proposed Shards system.

A framework for system construction does not suffer from the bootstrap problem. An immature framework is full of development opportunities. Early projects can define themselves in terms of the framework and gain extra value by becoming the foundations of future integrated systems. Project discoveries suggest future projects and iterative improvements to how projects can cooperate as part of a framework. At some point the bootstrap problem for the approach as a whole vanishes when the system becomes sufficiently mature it can be productively used in full system development efforts.

The idea of a framework for system development provides a solution to the original problem that drove the development of this thesis. Adding a significant contribution to the field of operating systems involved large projects and competition with evolved, sophisticated and mature systems. The combination of project scale, significant risk of project failure and difficulty in predicting a meaningful advantage over existing systems discouraged research. The solution is to create an environment where a small research project can gain direction and value from being a contribution to a larger development effort. The Shards system allows a broad range of research contributions developed independently to cooperate in the construction of future systems. The specifics of the future systems do not need to be known if there is a process for building systems from pieces. A process for building systems may find that increased capability and efficiency in system construction enables new opportunities in the creation of customised systems.

10.4 Future Work

As the previous section discussed, the Shards system is an approach and not a solution. It provides a framework for integrating advances in a very broad range of systems fields into a single process. The amount of future work it could lead to is thus very large.

The expressive power of the Shards system is directly related to the number of filters available (assuming that there is no duplication of functionality). The creation of a filter represents both an extension of the Shards system and a precise formulation of a practical operation. The more operations that are available, the more functional combinations that can be constructed. In a more practical direction, more filters mean it is more likely required functionality can be reused directly from a pre-existing filter. Filter development will also drive improvements in techniques for capturing OS mechanisms, shared ontologies and sophistication in the facilities offered by the underlying Shards process framework.

Existing filters are also a subject worthy of investigation in themselves and well suited to a solo individual project once the framework is defined. This is because they represent both implementations and theoretical models. Being able to determine functional equivalency or even functional superiority between two filters is a meaningful outcome. It indicates that two approaches, which probably have their own notations, names and descriptions are doing precisely the same thing. This determination is not only part of reducing duplication amongst filters but it also represents the discovery of unrecognised commonality in the field. A functionality that is sufficiently common can be taken as an indication of a general mechanism or principle.

Techniques for analysis and resulting optimisations are currently practical and popular subjects for investigation. The creation of a common framework could make this work more widely applicable. Many analysis projects are either abstract or strongly bound to a particular system. This means there can be significant complexity in judging the potential advantages and integration effort of including them in a new operating system. If they could be expressed as Shards filters, then integration and basic testing of their effect could become much more efficient.

The final milestone of maturity would be reached when a complete operating system can be completely expressed and optimised to project goals within the Shards system. This would act as an extremely convincing demonstration of two things. The first is that the Shards systems is expressively powerful enough to describe a complete operating system. The second is that the coverage of the filters is complete in that there is at least one filter for every operation contained in a mature operating system.

The amount of work this involves is safely beyond any single individual's capacity. It would represent many years of work by a specialist team and would undoubtedly give rise to many discoveries, challenges and innovations along the way. It is precisely this sort of scaling problem

that encouraged the development of Shards as an entirely modular system. However, if it was ever completed, it would represent a significant advance, not only for Shards, but also for the field of operating systems theory and education.

The generation of a complete operating system is likely to begin through taking parts from existing operating systems. Operating systems contain a wealth of mechanisms and design decisions that could be investigated as small and self contained studies. The challenge would be how the parts of the existing system could be expressed as Shards modules and reusable insights for future systems. The research papers published extend the understanding of the studied component and the role it plays in the wider system as well as adding to the Shards filter collection. Eventually, when the number of subsystems reaches a critical mass it then becomes clear that most of the pieces for a system already exist and it is unification that is required to make this evident. This approach more closely matches human behaviour than the idea of consuming an entire operating system in a single project.

In a similar fashion the growth of Shards would require a number of projects adapting components to specific project design goals, hardware platforms and system environments. The process of creating new derivations of base components and automating the the process of optimising them to a local environment and project needs is far from trivial. The process can be expected to provide some challenges for the Shards model but also a discussion on how many fundamental components exist and to what extent they prove optimal in specific cases.

Once a base system is possible, the focus could move to variations and specialisations. Different hardware platforms, heterogeneous and distributed systems, different core mechanisms and optimisation goals are all potential drivers for investigation. This process of investigation then drives extension of the Shards system. New capabilities increase the range of applications for which Shards driven generation and optimisation of a customised solution are commercially viable.

The vision for a fully matured Shards system is one in which any potential system could be expressed using Shards. This would allow discussion, analysis, comparison and experimentation to be performed at the level of the Shards project file. The project file would take the place of large amounts of documentation with much of the detail existing only in the source code as is common with systems now. When a researcher can offer a Shards project file to someone not involved in the project and that person could construct the system from that blueprint. When

the recipient could experiment with different applications, easily swap in new parts and express the new variant system in the same form. Then Shards would have become a true system of systems and a common language for their expression.

Bibliography

Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13. ACM SIGARCH and IEEE Computer Society TCCA, June 1995.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice–Hall, second edition edition, 1996.

Ada Augusta and Luis F. Menabrea. Sketch of the analytical engine with notes upon the memoir by the translator ada augusta, 1842.
URL: `http://www.fourmilab.ch/babbage/sketch.html`.

Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice–Hall, 1987.

Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 197–210. USENIX Association, June 2000.

John W. Backus. The IBM 701 Speedcoding system. *Journal of the ACM*, 1(1):4–6, January 1954.

John W. Backus. Programming in America in the nineteen fifties - some personal impressions. In *Proceedings of the International Conference on the History of Computing*, pages 125–135, June 1976.

John W. Backus and William P. Heising. Fortran. *IEEE Transactions on Electronic Computers*, 13:382–385, August 1964.

Jonas Barklund and Robert Virding. Erlang 4.7.3 reference manual. Technical Report Draft (0.7), Ericsson Telecom AB, 1999.

Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review (OSR)*, volume 29(5), pages 267–284. ACM SIGOPS, December 1995a.

Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, 1995b.

Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The pure family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC*. IEEE, May 1999.

Owen Bishop. *Exploring Forth*. Granada Technical Books, 1984.

Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the Usenix Summer 1994 Technical Conference*, pages 87–98. Usenix Association, June 1994.

Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, anniversary edition, 1995.

Frederic P. Brooks, Jr. No silver bullet—essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946). In *The Origins of Digital Computers: Selected Papers*, pages 399–413. Springer-Verlag, third edition, 1982.

Michael I. Bushnell. The HURD: Towards a new strategy of OS design. *Free Software Foundation, GNU's Bulletin*, January 1994.

Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placemenet. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, volume 33(11), pages 139–149. ACM SIGPLAN, November 1998.

Roy H. Campbell and See-Mong Tan. Choices: An object-oriented multimedia operating system. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 90–94. IEEE Computer Society Press, 1995.

Maurice Castro. Ec: an erlang compiler. Technical Report SERC-0128, Software Engineering Research Centre, 2001.

Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105. ACM, 1982.

Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, University of New South Wales, May 2003.

David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 179–194. USENIX Association, November 1994a.

David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX, November 1994b.

Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 13–24. ACM Press, May 1999.

Douglas Comer and Timothy Fossum. *Operating System design: Volume 1, the XINU approach.* Prentice–Hall, 1988.

Fernando J. Corbato and Victor A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the Fall Joint Computer Conference*, volume 27, pages 185–196. American Federation of Information Processing Societies, June 1965.

Fernando J. Corbato, Jerry H. Saltzer, and Charlie T. Clingen. Multics – the first seven years. In *Proceedings of the Spring Joint Computer Conference*, pages 571–583. American Federation of Information Processing Societies, May 1972.

Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233. IEEE Computer Society TCCA and ACM SIGARCH, May 1999.

Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

Peter Denning and Robert Brown. Operating systems. *Scientific American*, 251(3), September 1984.

Peter J. Denning. The Working Set Model for Programs. *Communications of the ACM*, 11(5): 323–333, May 1968.

Peter J Denning. Before memory was virtual. In *In the Beginning: Recollections of Software Pioneeers*. IEEE Press, 1997.

DexOS. Dexos website, 2011.
URL: `http://www.dex4u.com/`.

Chris DiBona, Sam Ockman, and Mark Stone. *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, 1999.

Edsger Dijkstra. The structure of the THE-multiprogramming system. In *Communications of the ACM*, volume 11, pages 341–346, May 1968.

Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, 1999.

DoJ:1999. Findings of fact, 1999.
URL: `http://www.justice.gov/atr/cases/f3800/msjudgex.htm#iiid`.

Dale Dougherty and Arnold Robbins. *sed and awk*. O'Reilly, second edition edition, 1997. ISBN 1-56592-225-5.

DR. *CPM 1.4 User Guide*, 1978.
URL: `http://www.gaby.de/cpm/manuals/archive`.

John Presper Eckert. Presper eckert interview. Technical report, Smithsonian Institution, 1988.
URL: `http://americanhistory.si.edu/collections/comphist/eckert.htm`.

The Economist. The other bill: Profile of bill joy. *The Economist, print edition*, September 2002.

Dawson R. Engler. *The exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999a.

Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999b.

Dawson R. Engler and Frans Kaashoek. Exterminate all operating system abstractions. In *Fifth Workshop on Hot Topics in Operating Systems*, pages 78–85. IEEE Computer Society Press, 1995.

Robert M. Fano and Fernando Corbato. Time sharing of computers. *Scientific American*, 215 (3), September 1966.

Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think: A software framework for component-based operating system kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002.

Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–152. USENIX Association, 1996.

Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for Kernel and language research. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Operating Systems Review, pages 38–51. ACM Press, 1997.

Alessandro Forin, Johannes Helander, Paul Pham, and Jagadeeswaran Rajendiran. Component based invisible computing, 2001. IEEE Real-Time Embedded System Workshop.

Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. ISBN 0-8053-1670-1.

Eran Gabber, Christopher Small, John Bruno, Jose Brustoloni, and Avi Silberschatz. The pebble component-based operating system. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, June 1999.

Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation: (OSDI '99)*, volume 32(5) of *Operating Systems Review*. USENIX, 1999.

Simson L. Garfinkel and Hal Abelson. *Architects of the Information Society: Thirty-Five years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999.

Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. ACM SIGOPS European Workshop, 2000.

Mel Gorman. *Understanding the linux virtual memory manager*. Prentice–Hall, 2004.

Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.

Andreas Haeberlen, Christian Ceelen, Espen Skoglund, Gerd Lieflander, Jochen Liedtke, Kevin Elphinstone, Marcus Volp, Uwe Dannowski, and Volkmar Uhlig. The L4Ka vision. Technical report, University of Karlsruhe, 2001.

Elliotte Rusty Harold and W. Scott Means. *XML in a nutshell*. In a nutshell. O'Reilly & Associates, Inc., second edition, 2002.

Robert Harper and Peter Lee. Advanced Languages for Systems Software: the Fox project in 1994. Technical Report CMU-CS-94-104, Carnegie Mellon University, 1994.

Grace Murray Hopper. The education of a computer. *Annals of the History of Computing*, 9 (3–4):271–281, 1987.

M. Elizabeth C. Hull. Occam - a programming language for multiprocessor systems. *Computer Languages*, 12:27–37, 1987.

I2O:1997. *Intelligent I/O (I2O) Architecture Specification*. I2O Special Interest Group, March 1997.

IBM:2002. Ibm eserver zseries 800 and z/os, z/os.e, z/vm and vse/esa reference guide. IBM Sales Manual, December 2002.

IEEE. *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE, 1990. ISBN 1-55937-061-0.

Itanium2hw:2002. *Intel Itanium 2 Processor: Hardware developer's manual*. Intel, 2002.

Itanium2rm:2004. *Intel Itanium 2 Processor: Reference Manual, For Software Development and Optimization*. Intel, 2004.

Intel64:2014. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2014.

Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *ACM SIGPLAN Notices*, 30(8):179–188, August 1995.

Margret Johnston. Microsoft tightens the vise on desktop OS market. *Infoworld*, February 2001.

JTF:2013. *Computer Science Curricula 2013*. The Joint Task Force on Computing Curricula, ironman draft edition, 2013.

Joachim Kempin. Trial document 365: email to bill gates, 1998.
URL: `http://www.usdoj.gov/atr/cases/ms_exhibits.htm`.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice–Hall, second edition edition, 1988.

Donald E. Knuth. Von Neumann's first computer program. *ACM Computing Surveys*, 2(4): 247–260, December 1970.

Donald E. Knuth. *MMIX: The art of computer programming*. Addison–Wesley, 2000.

Donald E. Knuth and Luis T. Pardo. The early development of programming languages. In J. Belzer, A. G. Holzman, and D. Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 6, pages 419–493. Marcel Dekker, 1977.

Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2k: A distributed operating system for dynamic heterogeneous environments. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society, 2000.

Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer-Verlag, 2003.

Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. *SIGOPS Oper. Syst. Rev.*, 40:133–145, April 2006.

Matthew Langham and Carsten Ziegeler. *Cocoon: Building XML applications*. New Riders, 2002.

R. LaRowe, J. Wilkes, and C. Ellis. Exploiting operating system support for dynamic page placement in a NUMA shared memory multiprocessor. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

Doug Lea. A memory allocator, 1996.
URL: `http://gee.cs.oswego.edu/dl/html/malloc.html`.

Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. *ACM SIGPLAN Notices*, 35(11):105–116, November 2000.

John R. Levine, Tony Mason, and Doug Brown. *Lex and Yacc*. O'Reilly, 2 edition, 1992.

Steven Levy. *Hackers: Heroes of the Computer Revolution*. Penguin, 1984.

Jochen Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250. ACM Press, December 1995a.

Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Operating Systems Review, pages 237–250. ACM Press, December 1995b.

Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

John Lions. *Lions' Commentary on Unix 6th Edition*. Computer classics revisited. Peer to Peer Communications, 1996.

Harold Lorin and Harvey M. Deitel. *Operating Systems*. Addison–Wesley, 1981.

Peter W. Madany. Javaos: A standalone java environment. A white paper. Technical report, Sun Microsystems, 1996.

Kostas Magoutis, Jose Brustoloni, Eran Gabber, Wee Teck Ng, and Avi Silberschatz. Builing appliances out of components using pebble. In *Proceedings of 9th ACM SIGOPS European Workshop*, pages 211–216. ACM, September 2000.

Charles C. Mann. The end of moore's law? *MIT Technology Review*, May/June 2000.

Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

Ken Mayes. Arena processbase implementation and compliance. Technical Report CNC/2001/001, Centre for Novel Computing, University of Manchester, 2001.

Ken Mayes and James Bridgland. Arena - a run-time operating system for parallel applications. In *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, 1997.

John McCarthy. A time sharing operator program for our projected IBM 709. Technical report, M.I.T., Computation Center, January 1959. Letter to the Director of the Comp. Center of M.I.T.

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.

Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, 1995.

Ron Morrison and Dharini Balasubramaniam. A compliant persistent architecture. *Software, Practice & Experience 30 (2000), Special Issue on Persistent Object Systems*, 30(4):363–386, 2000.

Arthur L. Norberg. Software development at the eckert-mauchly computer company between 1947 and 1955. *Charles Babbage Institute for the History of Information Technology*, 2003.

Computerworld Online. Microsoft: Breakup would unleash 'chaos', hurt economy, May 2000. URL: `http://www.computerworld.com/news/2000/story/0,11280,44881,00.html`.

M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *11th International Symposium on Computer Architecture*, pages 348–354. IEEE Computer Society, 1984.

David A. Patterson and John L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann, second edition edition, 1998.

Thomas Petzinger, Jr. The front lines: History of software begins with the work of some brainy women. *Wall Street Journal (Eastern Edition)*, November 1996a.

Thomas Petzinger, Jr. The front lines: Female pioneers fostered practicality in computer industry. *Wall Street Journal (Eastern Edition)*, November 1996b.

Rob Pike. Systems Software Research is Irrelevant, 2000. URL: `http://www.cs.bell-labs.com/who/rob/utah2000.pdf`.

Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, third edition edition, 1992.

Todd A. Proebsting and Scott A. Watterson. Filter fusion. In *Symposium on Principles of Programming Languages*, pages 119–130, 1996.

Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, volume 29(5), pages 314–324, 1995.

Sanjay Raina. Virtual shared memory: A survey of techniques and systems. Technical Report CSTR-92036, University of Bristol, 1992.

Richard Rashid, Robert Baron, Alessandro Forin, David Goluband Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: a foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE Computer Society Press, 1989.

Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associatesu, 1999.

Eric S. Raymond. *The Art of UNIX Programming*. Addison Wesley, 2004.

Richard K. Ridgway. Compiling routines. In *Proceedings of the 1952 ACM national meeting*, pages 1–5, 1952.

Arnold Robbins. *User-level memory management in Linux Programming*. Prentice–Hall, 2004.

Laura Rohde. Windows dominates on the desktop, 2003.

Zik Michael Graeme Saleeba. *A Self-Reconfiguring Computer System*. PhD thesis, Monash University, 1998.

Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

Jerry H. Saltzer and Project MAC. INTRODUCTION TO MULTICS. Technical Report MIT/LCS/TR-123, Massachusetts Institute of Technology, Febuary 1974.

Peter H. Salus. *A quarter century of UNIX*. Addison-Wesley, 1994.

Soumen Sarkar and Craig Cleveland. XML based document transform applied to application software development projects, 2001.
URL: `http://citeseer.ist.psu.edu/sarkar01xml.html`.

D. Shasha and C. Lazere. *Out of Their Minds. The Lives and Discoveries of 15 Great Computer Scientists*. Copernicus, 1995.

Donald L. Shell. The SHARE 709 system: A cooperative effort. *Journal of the ACM*, 6(2): 123–127, April 1959.

Robert Shelton. *A Lock-Free Environment for Computer Music: Concurrent Components for Computer Supported Cooperative Work*. PhD thesis, University of Melbourne, 2011.

Joel Shurkin. *Engines of the Mind: The Evolution of the computer from mainframes to microprocessors*. W. W. Norton, 1996.

A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating system concepts*. Wiley, 9 edition, 2013. ISBN 780470233993.

Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley and Sons, sixth edition, 2002.

Christopher Small and Margo Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard University, 1994.

Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–54. Usenix Association, 1996.

William Stallings. *Operating Systems*. Prentice–Hall, fourth edition, 2001.

John A. Stankovic. VEST – A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2211:390–402, 2001.

Guy Steele, Jr. *Common* LISP: *The Language*. Digital Press, 2 edition, 1990.

Christopher Strachey. System analysis and programming. *Scientific American*, 215(3), September 1966.

Hiroaki Takada, Yukikazu Nakamoto, and Kiichiro Tamaru. The itron project: Overview and recent results. In *Proceedings of the Industrial / Experience Session and Invited Talks of the 5th International Conference on Real-Time Computing Systems and Applications*, pages 3–10, October 1998.

Andrew S. Tanenbaum. *Operating Systems: design and implementation.* Prentice–Hall, 1999.

Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, second edition, 2001.

Ken Thompson and Dennis M. Ritchie. UNIX programmer's manual. Technical report, Bell Laboratories, 1979.

Scott Thompson, Paul Packan, and Mark Bohr. MOS scaling: Transistor challenges for the 21st Century. *Intel Technology Journal*, 3:19, 1998.

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, Second Series*, 42:230–265, 1937.

Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29:128–170, June 1997.

USvMS. Text of complaint in u.s. vs. microsoft, 1998.
URL: `http://www.cnn.com/US/9805/18/federal.complaint`.

V2OS. V2os website, 2002.
URL: `http://sourceforge.net/projects/v2os/files/`.

John von Neumann. First draft of a report on the edvac. In *The Origins of Digital Computers: Selected Papers*, pages 383–392. Springer-Verlag, 1982.

W3CXML:2004. *Extensible Markup Language (XML) 1.0.* W3C, 2004. Third Edition.

Brian Warboys. The IPSE 2.5 project: Process modelling as the basis for a support environment. In *Proceedings of the First International Conference on System Development Environments and Factories*, pages 59–74. University of Manchester, Pitman, 1990.

Peter Wayner. Sun gambles on java chips. *byte.com*, 1996.
URL: `http://www.byte.com/art/9611/sec6/art2.htm`.

Gerald M. Weinberg. *The Psychology of Computer Programming.* Van Nostrand Reinhold Company, 1971.

Adam Wiggins. A survey on the interaction between caching, translation and protection. Technical Report UNSW-CSE-TR-0321, University of New South Wales, August 2003.

Brian Wilcox, Erann Gat, Larry Matthies, Reid Harrison, Richard Volpe, and Todd Litwin. Mars microrover navigation: Performance evaluation and enhancement. *Autonomous Robots Journal*, 2:291–312, 1995.

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1995.

Robert W. Wisniewski, Dilma da Silva, Marc Auslander, Orran Krieger, Michal Ostrowski, and Bryan Rosenburg. K42: lessons for the os community. *SIGOPS Oper. Syst. Rev.*, 42:5–12, January 2008.

Paul T. Withington. The lisp machine: Noble experiement or fabulous failure?, 1991.
URL: `http://pt.withy.org/`.

Appendix A

Appendix

This section documents the functions used within the implementation of the Shards system.

A.1 Atom.h

```
/* The atom system efficiently stores strings and maps
 * each string to a unique integer value */

/* Must be called before the atom sub-system is used. */
void atom_init();

/* Performs unit testing on this code and logs the results */
void atom_ut();

/* Adds the string argument to the store and assings it an index */
int atom_put(char* name);

/* Recovers the string associated with the given index */
char* atom_get(int index);

/* Tests whether the given name exists in the store and has the
 * provided index value. Returns DC_SUCCESS if both are true */
int atom_match(char* name, int index);
```

A.2 data.h

```
/*

This code provides some facilities for accessing data nodes. It does
this by

- supporting some internal data types used by the Shards system.
- providing support for "generic" data types defined at run time.
- providing a mechanism where it will build a string from a node
  whether it is an inbuilt type, a dynamic type or through handing
  it off to a module family for decoding.

Each tuple will have the family and type stored as atoms. These are
used to determine who owns them. If they are "internal,*" then they
will be defined here.

The third data item will be a block of data containing an initial byte
defining its type. If this byte has the value "generic" then the data
item can decoded using the functions in this code. It will be arranged
in a sequence of ID bytes determining the type of the next data item.

If the type is not generic then control will be passed to the code
determined by the family of the node. This external code will know
how to represent the data types used by it and any members of its
family.
*/

/* create a new data item */
dyndata* data_new();
void data_free(dyndata* data);

/* reset the read position to the first item */
void data_rewind(dyndata* data);
```

```
/* clear the data */
int data_clear();

/* log details on error */
int data_error(dyndata* data);

/* check if all data items have been consumed, returns 1 if so */
int data_end(dyndata* data);

// Dynamic Type Functions
/* extends the dynamic data item to contain a new item with the given type */
void put_charA (dyndata* data, char* add);
void put_int (dyndata* data, int add);

/* attempts to read the given type from the current point */
char* get_charA (dyndata* data);
int get_int(dyndata* data);

// External Functions
/* converts the dynamic data type into a human readable string */
char* data_string (char* buff, dyndata* data);

/* runs the unit tests on this code and logs results */
void data_ut();
```

A.3 dchain.h

```
/* This module represents the data chain and functions directly related
 * to its representation and manipulation. Any components that can be
 * internalised should be. */

/* Initialises internal data structures. This should be called
 * before processing starts. */
void dc_init();

/* Cleans up the data chain */
```

```
void dc_free();

// internal unit test
void dchain_ut();

/* filter navigation */
// get next filter
void* dc_filterNext();
// add filter as next
int dc_filterAdd(dyndata* name);
// add filter at end of filter list
int dc_filterAppend(dyndata* name);

// functions to change to order of filter execution
void dc_filterSkip(dyndata* name);
void dc_filterReverse(dyndata* name);
// functions to cancel progression through the data chain
void dc_filterAbort();
void dc_filterEnd();
void dc_filterRedo();

/* data navigation */
// get next data item
int dc_next();
// rewind data read head to first item
int dc_rewind();
// recover data
int dc_getFamily();
int dc_getType();
dyndata* dc_getBody();
// remove data (to indicate it is being manipulated, may be a marker).
// moves the read head forward
void dc_consumeData();
// place new data onto the chain (next dc_next() completes transaction).
int dc_emitData(int family, int type, dyndata* data);
```

```
// all emitted data is converted into a fold, use family = 0 if not specific.
int dc_fold(int family);
int dc_unfold(int family);
int dc_setFlag(int flag);
int dc_getFlag();

/* data hiding functions */
// hide or show various types of data (eg. "internal")
void dc_hideFamily(int family);
void dc_showFamily(int family);
void dc_hideType (int type);
void dc_showType (int type);
// reset to default (hides internal and system)
void dc_defaultHide();

/* data search functions */
// find specific values (can be called repeatedly)
int findFamily(int family);
int findType (int type);
int findCouple (int family, int type);
int findRange (int lower, int upper);
int findMatch (char* string);
// avoid specific values (can be called repeatedly, can use hide as well)
int skipCouple (int family, int type);
int skipRange (int lower, int upper);
int skipMatch (char* string);
// execute and reset
int dc_search();
int dc_searchReset();

/* environment data (stored as {internal | family, "env", data}
 where data is generally a (name,value) pair but may have more fields */
dyndata* dc_getEnv (char* name);
int dc_putEnv (char* name, dyndata* data);
// environment variable is family specific
```

```
dyndata* dc_getEnv (int family, char* name);
int dc_putEnv (int family, dyndata* data_item);
// remove the data value
void dc_clearEnv (char* name);
void dc_clearEnv (int family, char* name);
```

A.4 filter.h

```
// use this as a wrapper for system calls to filters, thus also an
// api that filters must supply.

typedef enum filter_result_t {
    FILTER_SUCCESS = 0,
    FILTER_ERROR
    } filter_result;

void* filter_open(char* name);

filter_result filter_exec(dyndata* arg);
```

A.5 global.h

```
#define OBUFF_MAX 1024

#define DC_SUCCESS 1
#define DC_FAIL 0

#define TRUE 1
#define FALSE 0
typedef char bool;
```

A.6 loadlib.h

```
/* These functions manage the use of dynamic libraries / filters */

// The directory in which to look for filters
```

```
void lib_setdir(char* dir);

void* lib_open(char* libname);

int lib_call(void* filter, char* cmd, void* arg);

void lib_close(void* handle);
```

A.7 log.h

```
/*
 * The idea is to mimic the behaviour of printf while having user selectable
 * outputs (including timestamped file) and selectable priority levels. Also
 * do enter / leave macro's that use the C precompiler defines.
 */

// most functions are no-ops when not in debug mode
#ifdef DEBUG
#define log_dbg(a...) print_log(LVL_DBG, __FILE__ ,LOG_LOCATION,## a);
#define log_warn(a...) print_log(LVL_WARN, __FILE__ ,LOG_LOCATION,## a);
#define LOG_ENTER print_log(LVL_DBG, __FILE__ ,LOG_LOCATION,"Entering Function");
#define LOG_EXIT print_log(LVL_DBG, __FILE__ ,LOG_LOCATION,"Exiting Function");
#else
#define log_dbg(a...)
#define log_warn(a...)
#define LOG_ENTER
#define LOG_EXIT
#endif

#define log_err(a...) print_log(LVL_ERR, __FILE__ ,LOG_LOCATION,## a);

#define log_lock() log_lock_int(LOG_LOCATION)

// Open the log file
void log_openfile();
// log only items of this priority level and higher
```

```
void log_set_priority(int level);
// restrict logging to given function name only
void log_lock_int(const char* name);
void log_unlock();
// close the logfile
void log_closefile();
```