PARALLEL I/O USING A DISTRIBUTED DISK CLUSTER: AN EXERCISE IN TAILORED PROTOTYPING

Charles Falkenberg James M. Purtilo

Computer Science Department and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT: *Tailored prototyping* refers to an emerging process for prototyping software applications, emphasizing a disciplined experimental approach in order for developers to obtain an understanding of system characteristics before committing to costly design decisions. In our approach, the design of software constituting prototype apparatus is driven by experimental hypotheses concerning risk, rather than an application's functional requirements. This paper describes the principles behind tailored prototyping, then illustrates them in concrete terms by describing their application in a pilot project. The pilot used in our illustration is a parallel I/O service — a mechanism designed to deliver pages, in parallel, from a cluster of distributed disks. The performance results show that this parallel I/O system can, in certain circumstances, deliver higher page throughput from multiple remote disks, than with a single local disk. The pilot project exemplifies our prototyping method which is applicable to a wide variety software prototyping activities.

INDEX TERMS: Software prototyping, parallel file systems, software risk analysis, distributed systems, database management systems, I/O bottleneck.

With oversight by Office of Naval Research, this research was supported by ARPA in conjunction with the Prototech project.

1 INTRODUCTION

When a setback occurs during the development or installation of a large software system, it often seems obvious that effective prototyping could have prevented the problem. Before the fact, however, identifying those aspects of the system or installation which will be problematic, and understanding the actual cost and benefit of prototyping is not obvious. The benefit is not usually appreciated until the scope of problem is uncovered and the cost can be excessive unless prototyping activity is optimized. The difficulty is knowing what to prototype, and when it will be cost effective.

As the design of a large system progresses, risk assessment and analysis can identify when a choice between strategies is critical. If the design options pose a high risk along with a high potential return, a prototype may steer the decision or, if development continues, build confidence in the choice. The result of the prototype is the validation or invalidation of a hypothesis that is the basis of the choice. Optimally, the prototyping activity should be tailored to this hypothesis, and constrained to issues and experiments which will reduce the risk of a given course of action.

Using our tailored prototyping method we undertook such an activity. In order to build confidence in the viability of a parallel I/O system, we constructed a running prototype of the system and performed experiments that refined the operational environment in which such a target system could provide an improvement in I/O throughput.

The larger design effort was the development of the ADMS database management system at the University of Maryland. The design and development of ADMS is a long term research project to build a comprehensive database management system based on the the concept of incremental updates [RES93]. Like any data intensive application, the developers of ADMS seek to minimize the I/O bottleneck. The critical decision was whether or not to develop a parallel file server to deliver pages to an ADMS client. A working hypothesis was formed by the designers that a cluster of remote disk servers would incur disk latency in parallel and deliver pages to a client at rates which approached the speed of the network with which they were connected.

To validate this hypothesis we, along with the developers of ADMS, believed that a working prototype was needed to provide constructive proof of the such a system. In addition, the construction of this prototype would expose the potential weak points of this design alternative and provide a test bed for the experimentation. The experiments would also quantify the hypothesis and refine the environment in which this type of I/O system could deliver improved performance.

In this paper we discuss the nature of the tailored prototyping process and how it was applied to the development of this prototype. We present the formation of the prototyping questions and the resulting requirements for the prototyping apparatus. We also present the problems and motivation behind the parallel I/O system and the experiments which led from the prototyping questions to the parameters for a successful implementation. Our results refine our definition of tailored prototyping and provide a performance envelop for this type of parallel I/O system.

In the next section we describe our approach to prototyping and present our methodology. This is followed in section 3, by a brief description of ADMS along with the risks associated with the proposed parallel I/O system. Section 4 describes the application of our methodology to the design and development of the I/O prototype which implements an object called Pagefiles. The performance experiments are presented in 5 along the conclusions we draw from the experiments and in the last section we evaluate the success of the prototyping activity and how it refined the tailored methodology.

2 TAILORED PROTOTYPING

The goal of prototyping is to model a target system in order to collect information that will clarify or reduce a risk. The way in which the prototype models the target system can vary. The prototype may be an early, alpha, release of a software product or it may model a very small or abstract part the target system. The information gained from the prototyping exercise should validate (or invalidate) a key concept, design, or a development strategy and therefore increase the likelihood of a successful implementation of the target system. The primary result of prototyping is the information gained and the usefulness of the prototyping apparatus to later products is usually secondary.

Recently, the variety of prototypes have been successfully classified by several authors ([WK92, LSHZ93]). These classification schemes distinguish between temporary or "throw away" prototypes and those prototypes which become the building blocks of the target system. We will adopt the classification system of [WK92] and call these temporary prototypes conceptual prototypes. These are distinct from evolutionary prototypes which become part of the target system. In the extreme case software maintenance becomes a series of evolutionary prototypes which provide information for a subsequent release.

Software systems built in a research environment are most often conceptual prototypes. The main purpose is to reveal problems and demonstrate properties about the target system. If the prototype is not thrown away it is kept as a foundation system for a future prototype of an enhancement. The prototype, however, rarely becomes part of an operational system which is maintained for a broad distribution. The goal of research is to expand understanding and not to produce a product which is marketable. Software research prototypes, therefore, fall naturally into this class of conceptual prototypes.

In this paper, we do not address in detail the issues of defining and classifying prototypes. Rather, we are concerned with the methodology of prototyping. We focus on the iterative nature of the prototyping activity which is common to most software prototyping. A risk is identified and a prototyping experiment is designed address the problem. The results of the experiment are then used to reevaluate the risks before new experiments are designed. In our methodology, risks are prioritized and measurable questions are established which will, when answered, reduce the highest priority risk.

Tailored prototyping is a methodology which outlines the steps involved in each iteration of the discovery process. The emphasis of this method is to quantify the value of the results, as well as the costs, of prototyping in order to manage expectations and provide a yardstick for success. Each iteration is made up of the steps outlined below.

1. **Identify risks associated with a critical design choice**: Any major design alternative which makes assumptions about an untried technique or technology poses risks. The intent that the design alternative will meet the functional and performance requirements is based on several of these assumptions or hypotheses. This step identifies the risks associated with the invalidation of those assumptions. The risks include both the probability of failure and the severity of the consequence of failure.

2. **Formulate questions**: Establish a minimal set of questions which, when answered, will prove or disprove the validity of the critical hypotheses.

3. **Tailor the prototype**: Prioritize the questions a consider the experiments needed to answer those questions. Select a prototyping mechanism and tailor it to those experiments.

4. **Evaluate cost**: Estimate the cost of the necessary prototype and compare this with the risks of invalidating assumptions. This must include some preliminary design.

5. **Construct the prototype apparatus**: The prototype can be broadly interpreted as instrumentation in existing systems, a design walk through with potential users, or a specially constructed piece of software which models the some part of the target system.

6. **Execute the experiment and analyze the results**: Once the experiment is performed examine the results and answer the prototyping questions

7. **Incorporate the experimental results**: The goal is to produce information and not, necessarily, functioning code. Examine the prototyping questions in light of the new experiments and re-evaluate the risks.

The elements of risk include the probability of failure and the severity of the consequence of failure. When the 'risk of failure' is discussed it is the union of these two. An option is considered risky if there is a high probability that an alternative will not succeed, even if the consequence of failure is not great. On the other hand, if the probability of failure is small but the severity of the consequence is high, minimizing this probability can be well worth the effort. Either factor will increase the overall risk of failure.

We refer to tailored prototyping as a methodology, not just a method – the particular steps taken by a developer using our approach may well appear very different from project to project. Instead of turning to a cookbook of off-the-shelf prototyping recipes, the developer must plan how the methodology will become manifest in his own particular situation. Many scenarios do indeed have much in common for certain of the steps, and hence we have found some tools useful for assisting the developer [CPP94]; and because of the extent of planning, and also potential for leveraging prior experiences once captured, it is easy to anticipate how in the future emerging decision support systems may support broad classes of applications employing our process. Nevertheless, the heart of this process is planning: that's the point.

Our simple expression of tailored prototyping in terms of seven steps is the result of observations made over a broad class of applications and domains, wherein developers were denied success in prototyping ventures. Rather than state these examples (which were the basis for this work) 'in the negative', we have found it useful to summarize the experiences as aphorisms associated with each step, to guide future prototypers:

1. Step one: *Know why you are prototyping in the first place.* One of many bases for prototyping is of course to discover requirements, and it is therefore understandable that considerable uncertainty precede a development activity. But when faced with uncertainty together with an unguided mandate to "build something," programmers will naturally build what they know – when in fact the purpose should be focus on what is not known. We have observed too often developers have been content to build the 'easy' parts of an application, and code around dangerous issues in the prototype, only to later be sunk by what had been ignored earlier.

2. Steps two and three: *State up front how you will recognize when you have succeeded in prototyping*, then *make sure what you build relates to your reasons for prototyping.* Of course, the success criteria should be expressed in a testable form. Developers should not, for lack of planning, implement more functionality than is necessary to address their objective hypotheses. Well-intentioned developers often have ideas for interesting new 'features' that might be installed in a growing prototype, but without objective statements of what is required of the apparatus, there is no way to measure whether that new feature ought to be added.

   An important aspect of these maxims is in how they help developers know when to *stop* prototyping – when the success criteria has been satisfied or denied. All too often we have seen developers continue a prototyping effort without much understanding of whether they are making progress; after achieving some basic execution capability, they keep adding features until either the budget forces a stop, or until the manager arrives to declare that the prototype has just been promoted to product. "Ship it!" Since calling software a "prototype" has traditionally been a magic incantation to ward off obligations for configuration control, documentation or regression testing, promotion of a prototype to product is tempting failure. By focusing on concrete and testable objectives for the prototype, developers should have an easier time explaining to the above manager what is the difference between prototype and product.

3. Step four: *Understand the cost of prototyping and planning— its an investment.* We have observed a wide belief that prototyping is somehow supposed to be cheaper than making a product (then it gives the same results.) This is true only if one looks at the life cycle of a product; it is not necessarily true if one considers early development costs. Tailored prototyping actually might introduce greater costs on the front end; the payoff comes in fielding a higher quality product. In general, we have observed that early life cycle costs are

generally only low in processes where prototyping doesn't encompass configuration control, documentation, regression testing ... that is, in processes that sacrifice engineering rigor.

4. Step five: *Build only the apparatus you need.* This step is generally the easy one, once people know what to build as a result of the prior steps. It is also this step which people previously have focused upon, generally to the exclusion of other steps of this process.

5. Step six: *Remember to use your prototyping apparatus.* That might sound like a silly aphorism, for who would forget to run the programs once they built them? Plenty of developers, as it turns out. Too often, we have observed programmers focusing upon construction of the apparatus as the end result – instead of using that apparatus to obtain other information critical to success of the overall effort. After running the apparatus on a few easy test cases, the team moves on, when they should have pushed for rigorous testing. Projects which involved prototyping but which subsequently failed, have been analyzed to discover that the earlier prototype did indeed exhibit behavior warning of disaster, had only the developer either bothered to exercise the apparatus thoroughly, or had a basis (via planning, as put forth here) for recognizing the behavior as being a warning.

6. Step seven: *Use what was learned.* This is much the same advice as for step six, but it bears repeating. We have see situations where the useful information about a proposed product was indeed exposed by prototyping - but because of a breakdown in this abstraction step, the information was not incorporated into the product's design specification. Many researchers have observed how the key experiences of prototyping are too-often present only in the team's memory, which is then lost when the team moves on to other projects. We observe that when the objective prototyping hypotheses are captured as in tailored prototyping, so can the results of prototyping; the information can be recorded and used.

The tailored prototyping methodology is applied during the development or installation of a large software system. The development of ADMS at the University of Maryland is an example of this level of development. A proposed enhancement to ADMS to implement parallel I/O, provided an opportunity to apply the tailored methodology, explore the value of prototyping, and assist in refining the design of the target system.

3 DATABASE RESEARCH USING ADMS

ADMS is a database management system (DBMS), developed at the University of Maryland [RES93]. As a research tool, ADMS was designed to explore incremental access methods for caching query results and updating indexes. In addition, ADMS has been used to research database issues relevant to spatial data management and query execution probability.

Although this is a research tool the design and development of this system closely parallels that of a production system. Design decisions have long term consequences for the system in the way quite similar to that of a marketable DBMS. The fact that ADMS is a research tool does lower the aversion to risk that would be found in a DBMS company, however, for the purpose of evaluating our prototyping method we view the development of ADMS as we would the development of a similar large commercial system.

3.1 I/O Bottleneck

As with any data intensive application the developers of ADMS are interested in improving I/O performance. Although disk speeds and I/O throughput have improved steadily in the last 15 years they have not kept pace with the dramatic improvement in processor and network speeds. This difference has been a problem for some current systems and will be a larger problem for new applications which have increased demand for high volume throughput. The performance of current database systems is often bound by the amount of data that can be read in and written out to the disk. New types of applications, like image processing and GIS, will place even higher I/O demands on database management systems.

Compounding the problem of disk latency, large disks and the high degree of concurrency found in most database applications results in a type of false sharing. Multiple requests for different pages, on the same disk, are queued up while the current request is being served. Even if there is no logical conflict with the requests, they must be serviced sequentially. The disk is essentially locked while the current request is being served.

These problems have prompted research into parallel I/O systems which attempt to overcome disk latency problems by incurring the cost of multiple I/O requests in parallel. Although several different parallel I/O systems have been researched, most of them depend upon a specialized file system or operating system. The developers of ADMS proposed a architecture which could take advantage of the existing UNIX operating system and file system and provide a parallel I/O cluster with off the shelf hardware.

3.2 A Tower of Pizzas

The architecture proposed by the developers of ADMS spreads disk pages across a cluster of workstations. When a client makes multiple requests for disk pages, the latency associated with accessing each page can be incurred by different disk drives in parallel. If the disk cluster is connected to the client machine with a network, the average cost of delivering a page to the client could be closer to the network latency than to disk latency. Figure 1 shows the initial concept of the system. The stack of disk servers is made up of the main enclosure, or pizza box, of a workstation. If multiple clients are requesting pages which are located on separate servers in the cluster, then different clients might be serviced simultaneously and the false sharing is reduced.

The central hypothesis for this system is that if disk latency could be incurred in parallel then pages could be delivered to a client at near network speeds. This has great potential because although disk access times can range from 20ms to 40ms, the round trip time for a message

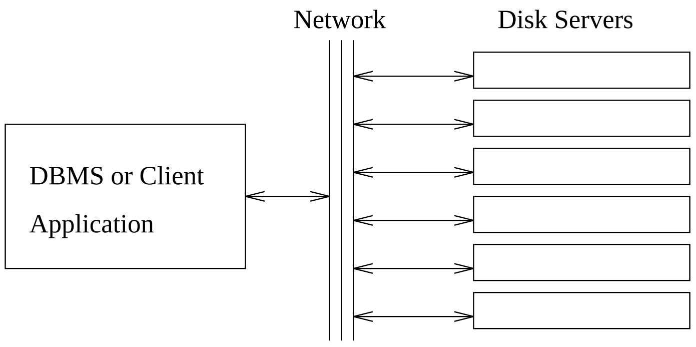

Figure 1: Stack of disks serving a single client over a network.

on current Ethernet networks ranges between 2ms to 4ms. If the hypothesis holds a marked improved in throughput could be realized.

There are several other potential advantages to this architecture. The distribution of pages could be optimized, balancing the requests for hot pages across all servers. Pages might be mirrored in order to provide fault tolerance and perhaps added parallelism. Finally, pages distributed on several disks could be accessed by different clients without interference. Figure 2 shows this multiple client architecture in which clients, connected to the cluster, would have independent access to pages on different disks in the cluster.

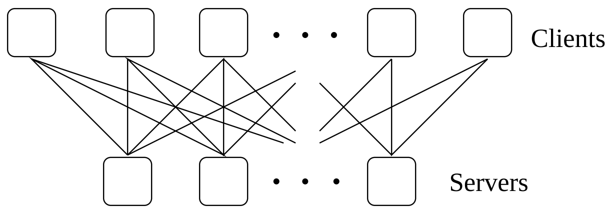

Figure 2: Conceptual configuration of multiple clients and a cluster of servers.

This multiple client configuration is potentially scalable. As more clients are added, the throughput for each client may be maintained by increasing the number of servers in the cluster. This is a long term goal and the scalability, and even the viability, of this type of parallel I/O system depends on the saturation point of the network under these conditions.

The potential advantage of this design is great but several of the underlying assumptions are unproven. The starting point of a tailored prototyping method is the analysis of the risks posed by these assumptions. The assumptions themselves must be understood before the questions necessary to prove or disprove these assumptions can be formulated. These questions will optimize the prototyping process by restricting the scope of the prototyping apparatus and the scope of the prototype should always be understood before the decision to prototype is made.

3.3 Risk Assessment

The fact that this architecture is untried, and the underlying assumptions have not been confirmed, increase the probability that this design would not provide all of the intended benefits. To understand the exposure to risk we must, therefore, understand the assumptions on which our design choice or hypothesis is based. These include assumptions about performance of the network and the disks; assumptions about effect of the architecture and our ability to use a specialized API; and lastly an assumptions that the additional overhead needed to control the system will not consume the benefit. The likelihood that not all of these will hold, shakes our confidence that this system could be installed and the benefits realized.

First, our hypothesis assumes that we can achieve the reported network speeds for large page sizes. If this is true we are also assuming the network speed will not diminish under the volume of requests we expect. Secondly, the power of the architecture is based on the distribution of work across multiple servers. We can only achieve network delivery speed if the disk expense can be incurred in parallel. We are assuming, then, that we can generate this level of concurrency with a standard work load. This means that any specialized API must be exploitable by ADMS. Finally, we are counting on the fact that the administration of the pages and requests will not add an undue overhead to the system.

These assumptions are risky and the success of the system is based on all of them proving valid. In addition, the fact that the system does not have much precedent means that even after the design phase, many unforeseen problems might occur which could jeopardize a successful implementation.

The size of the development effort can be translated into the extent of the risk. The failure of a large project poses much greater risk than the failure of a small piece of that project. But if a larger project depends on that small piece or if the design decision will restrict the viability or flexibility in the long term, even a seemingly small choice could pose large risks.

In the case of ADMS, a system which did not meet expectations would still have the benefit, albeit small, of advancing DBMS research. But the development time would still be lost along with the potential for some positive results. If a full scale commercial implementation of this system were to fail, not only would significant development time and resources be lost but the long term, strategic, planning would be invalidated as well.

In a commercial setting, even a successful implementation of a design alternative commits the company to a direction which may prove unwise in the long run. In this example, a commitment to this architecture poses serious ramifications for several other critical database concerns. Distributed page locking, crash recovery, and distributed page cashing are all problems which must be addressed in the overall design of the system. A commitment to this cluster architecture will dictate much of the design of these requirements, increasing the development cost and the cost of failure.

The severity of the consequence of a full scale implementation not providing the expected performance results, combined with the apparent simplicity of the tests needed to build confidence in the potential performance of the target system, compel us to consider a prototype. Without committing to the prototyping activity, the prospect of confirming our assumptions, providing experience with the target API, and perhaps building a test environment which could be used to examine the larger design consequences is sufficient reason to pursue the possibility of prototyping.

4 A TAILORED PROTOTYPE OF PARALLEL I/O

The tailored prototyping method starts with the assessment of risks attached to the target system. This is done as part of the design or development stage but a certain point, as the degree of risk increases, the option of prototyping needs to be considered. At that point the specific risks need to be turned into answerable questions. These questions are converted into experiments, and, if it is cost effective, a prototype apparatus is created which can run the experiments. Once the questions are answered, our understanding of the system improves or new risks may be identified.

4.1 Formulate Questions

Our initial goal with this prototype is to confirm the expected performance improvements. Subsequent development of the target system or continued analysis of the prototype is predicated on quantifying the performance of the target system. We therefore focus our questions on the issues surrounding the central performance hypothesis that a parallel I/O system can deliver pages at near network speeds.

Drawing from the assumptions on which the hypothesis is based we have several central questions:

1. Will concurrent disk operations provide throughput which approaches network throughput, and how does this compare with throughput using a locally attached disk? Answering this question will validate or invalidate several other assumptions we have made about the performance of the network under the data load we expect.

2. What is the API necessary to exploit this parallelism and can existing, data intensive, applications take advantage of it? This question is based on the concern that the new system might place special requirements on the applications which use it. The general usability of the API must be understood if the success of the system is to be evaluated.

3. How much overhead will be devoted to the administration of the system and how complicated will the target system need to be? This questions addresses efficiency of the system but more importantly it addresses the problem of building a new, untried, system from scratch.

4. What is the maximum throughput we can expect from the target system and how much does it depend on the performance of the network? This architecture provides disk parallelism, but the network is still a shared resource and we expect it to be the limiting factor. Without quantifying this relationship between the system performance and network performance we can't make judgements about the future potential of the system.

5. Will this architecture be scalable for multiple clients? In other words, as the number clients increases, can throughput levels be maintained by increasing the number of servers? This question is predicated on the fact that the throughput for a single client is high enough.

6. How will page locking, crash recovery, and page caching be accomplished with this design? These larger design issues will not be addressed here but the need to answer them will effect the choice we make in prototyping.

7. What are the other long run issues of committing to this architecture? This is a management question which cannot be answered directly by a prototype. It is significant, however, and it will effect the choices that are made, and the information that is sought during prototyping.

This is a broad range of questions which need to be answered before large scale development of the target system is undertaken. They are prioritized with resect to the activity of prototyping. The questions we can expect a prototype to answer are at the top, ordered by importance. We can be confident that if the set of assumptions we made initially is complete that the answers to these questions will be sufficient to build confidence in our design.

4.2 Tailor the Prototype

4.2.1 Select a prototyping mechanism The experiments necessary to answer these questions could take several forms. Discrete event simulation, constructive prototypes or even a preliminary design are all activities which will return information that could answer some of these questions. Simulation could provide some answers to the performance questions (1, 4 and 5). A design of the target system along with an applications which would use it, could answer question number 2 about the API. In this case, however, there is much that supports a constructive prototype.

A constructive prototype which passed disk pages over the network, and supplied a preliminary API to a test client, would provide not only a test bed for the current round of experimentation, but a foundation for extended experimentation of any advanced features of the target system. Performance figures which had been established by a running prototype would be more persuasive and the preliminary API could be designed, implemented, and then utilized by a testing program. In addition, a running system would also answer the question about the cost of page administration.

The impact of this design on locking and logging strategies (question 6) is not trivial. The design of these functions, however, will be a considerable investment itself, and it will only be warranted if the basic performance hypothesis is validated. If the initial results are promising, a constructive prototype would provide a valuable testbed to perform this additional prototyping work.

Finally, without also identifying all of the strategic issues, question 7 cannot be answered directly by any prototype. Some of the information needed to answer this question can, however, be provided by a prototype which uncovers the appropriate applications of this type of I/O system. A constructive prototype will, in addition, provide management with a example of the complexity necessary to implement the target system.

4.2.2 Design the experiments

Given a constructive prototype which implements a vertical subset of the target system functionality, we consider the experiments necessary to answer the highest priority questions. The experiments needed to answer questions 1, 4, and 3 will be straight forward to implement. To adjust the level of concurrency (question 1), the prototype must allow a varying number of page requests to be 'in process' at any given time. The size of the requested pages must also be varied in order to evaluate the potential volume of data which could be provided by the target system (question 4). To check the administration overhead, the performance of the prototype under the various loads must be compared against independent measures of I/O and network performance.

The issue of scalability is an important aspect of the target system. The prototype must be able to be modified to show the performance results for multiple clients and a varying number of servers. This experimentation is to be part of a second round of prototyping after some initial performance results are established. However, the flexibility of varying configuration in order to perform these experiments needs to be incorporated into the initial design of the prototype.

The development of a constructive prototype which implements an API for the target system will provide the answers to question 2. The experiment is the use of this new API by the testing program. The other questions have no direct experimental requirements other that to document the complexity of the system and identify the set of applications which could use the I/O system. With these requirements in mind we are able to design the prototype and estimate the overall cost of prototyping.

4.3 Estimate the Cost of Prototype

Our proposed prototype is a vertical implementation of the target system which will include those components needed to answer the prototyping questions. The prototype will read and write disk pages to a distributed cluster of machines connected to the client with an Ethernet network. Together with the developers of ADMS, we would design the API which could be used for the target system but the implementation should be restricted to the requirements of prototyping.

Evaluating the cost of the prototype is a critical step of the method. If the cost will be high, then the prototype itself is risky and the prototyping activity may not be warranted. Whether the prototype is limited construction of the target system or a simulation, the job of estimating the cost is difficult. The functional requirements need to be understood and enough of the detail design done to establish the scope of the project. At this point the cost of the prototype may prohibit its development.

After reviewing the system requirements, we estimated that one person, working about 1/3 time, could design the API and the prototype, build the prototype, and execute some preliminary tests in 3 to 4 months. In our case, building the prototype was an opportunity to experiment two other research topics. We wanted to evaluate the performance of our Polylith interconnection system and we wanted to experiment with the prototyping method itself. We would use the opportunity to refine the steps in the tailored prototyping process by building the prototype and examining the aspects which were not necessary to answer the important questions.

We proceeded with the prototyping activity because the cost was not significant and the potential for effective experimentation was. The following sections describe the requirements of the prototype we built and the experiments we ran using that prototype. Some of this design was done before the cost of the prototype was estimated.

4.4 Design Prototype Apparatus

The requirements specification for the Pagefile system were broken into two groups. The first set are functional requirements which apply to both the prototype and the target system. These included the API requirements and basic architectural topology. The second group of requirements apply only to the prototype. Examples of these include the instrumentation needed to gather statistics for the experiments and the internal interfaces needed for the prototype. Where appropriate, the design decisions which were made to meet the requirements are described in this section as well.

4.4.1 Functional requirements There were three central requirements for the prototype: the topology of the disk servers, the programming interface (API), and a model of the workload used for input. The system topology required that a variable number of clients access disk pages from a fixed number of disk servers. Each client is connected to the servers with a standard Ethernet network. The number of clients can change but a Pagefile is only valid for a single configuration of servers.

The prototype models a scalable parallel I/O system which needs to be accessible to a mid-sized installation. Therefore, the disk servers and network interconnect must be standard technology which is available to a wide variety of users. To achieve this, the disk servers are to be in the main enclosure of a workstation. This includes the CPU, memory and operating system but no keyboard or monitor. A stack of these enclosures will make up the cluster of disk servers [Rou92].

The prototype needed to provide a programming interface which could be used by an application program. The interface to a Pagefile closely resembles the UNIX file interface with a few distinct differences. First the granularity of an I/O request is pages, not bytes. Second, the read and write operations were each split into two steps; a request and confirm. This allows a server to operate asynchronously while the client is performing other work or making page requests to other servers. Finally, a Pagefile can also be stored on a disk which is local to the client and the designation of a Pagefile as local or remote is done only when it is opened. All other access is performed with a file descriptor which is unique across all open Pagefiles. With these exceptions the API performs the same set of operations available with standard UNIX files: `open`, `close`, `read`, `write`, `unlink`, `status`, and `lock`.

The last, central, requirement was that the prototype be able to operate with an actual DBMS workload. A sample workload was captured by ADMS and an application program, using the Pagefile API, was written to read in that workload and perform I/O on Pagefiles. This allowed the performance of the prototype to be compared with the identical execution of ADMS using serial I/O.

4.4.2 Prototyping requirements The second group of requirements originated from the definition of the experiments for which the prototype was to be used. These requirements tailor the prototype to meet the needs of the experiments defined as part of the methodology. These are not functional requirements, but requirements that allow the the final prototype to be instrumented in a fashion which provides for accurate experimental results.

The primary parameter of the experimentation is the topology of the Pagefile system. The topology describes the number of servers and clients and the interconnections between them. The effect of modifying other parameters, like page size or concurrency, will likely depend on the specific configuration. A flexible interconnection system is needed to allow the topology to be changed as the experiments proceed. In addition, a high level interface to the interconnection system had the potential of shielding the prototype from possible changes in the network implementation. For these reasons the Polylith interconnection system was intended to be an integral part of the prototype. In addition, we wanted to stress test Polylith and the high volume of requests, and the demand for high performance, provided an good opportunity.

As a prototyping tool the Polylith interconnection system is designed to provide for reconfiguration of application topology. The high level, message passing, interface to Polylith reduces the effort required to to develop the initial configuration, and change that configuration to meet the demands of the prototype.

The initial set of experiments evaluate concurrency, page size and local and remote throughput. The instrumentation needed to achieve this requires a test program and a set of timing functions. The functional requirements specify that location of a Pagefile, its page size and the number of 'in process' requests are established by the application utilizing the Pagefile system. This means that the instrumentation of these experiments will not be in the system itself but in the test program which exercises the prototype. The test program accepts the input parameters needed to govern each experiment.

The second tool was a simple execution timer which collects and saves the statistics of each execution. Although the collection of any particular statistic was not complex, the set of statistics which were collected evolved as the experimentation continued, making this one of the most volatile of the interfaces.

To compare the performance of a remote Pagefile, spread across the server cluster, with a Pagefile which was local to the client, a single local interface which was needed. This internal interface was used by the Pagefile client for local Pagefiles and by the Pagefile server for remote Pagefiles. This isolated the remote processing as the only additional variable in the statistics for remote throughput.

Finally, a high level definition of the interface to remote Pagefiles was needed to hide the implementation of the remote processing. Although POLYLITH provides a complete set of remote accessors it was isolated from the general processing in order to allow another interconnection architecture to be tried. The basic remote page file functionality does not depend on how the communication is implemented and so a standard set of remote Pagefile accessors provide independence from the implementation.

These last two requirements lead to the design of two internal interfaces. The Pagefile local interface (PFLI) is used at the client side to access local Pagefiles and by the server to fill requests from the client. The Pagefile remote interface (PFRI) encapsulates the access to remote Pagefiles. Figure 3 shows the relationship of these two interfaces along with the API which is was called Pagefile external interface (PFEI).

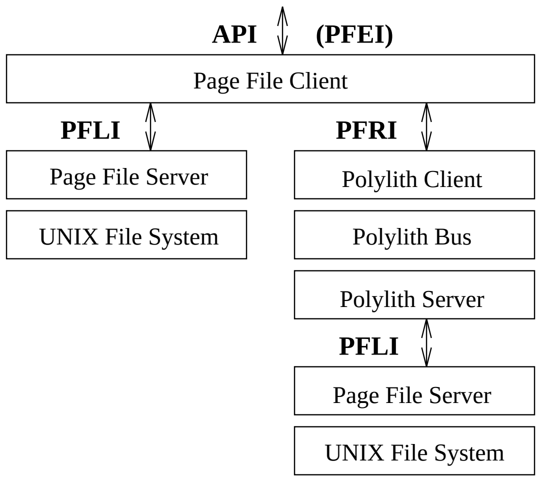

Figure 3: Interfaces defined for the Pagefile prototype.

4.4.3 Prototype Development During the first phase of development the details of the external interface or API were finalized with the developers of ADMS. The API included the standard operations on UNIX files and a call to initiate the communication with the cluster of servers as well as a call to terminate the system. The manual for the complete API is provided in [FHK93]. The local and remote interfaces were modeled after the Pagefile API which resulted in corresponding accessor in the remote and local interfaces for each accessor in the API.

Figure 4 shows the components of the prototype implementation. An application program, using the PFEI, links to the client side of the prototype. The prototype uses the POLYLITH system to connect to the servers for each disk. Each server operates independently on a separate workstation which could be devoted to file service.

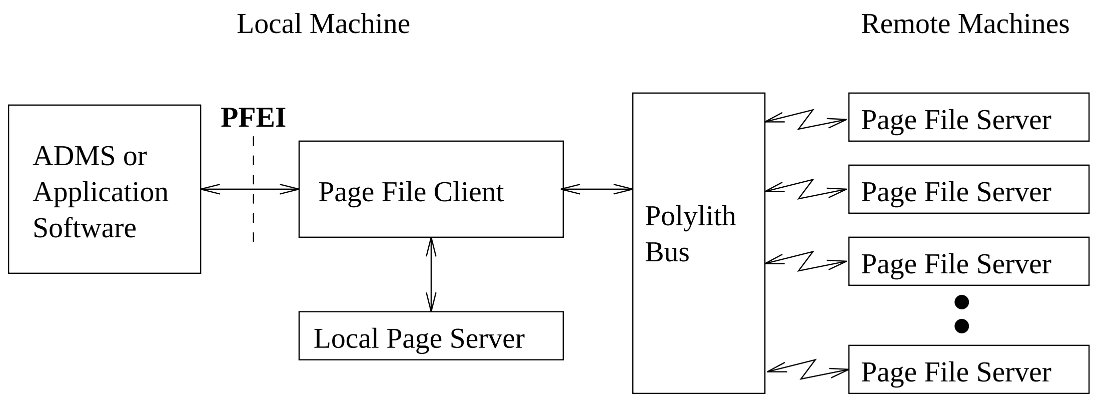

Figure 4: Architecture of Pagefile prototype.

When the system had been successfully implemented and tested a prototyping driver was designed and developed. This driver represented the application code which would make use of the Pagefile system. Unlike an actual application, however, it read in a formated file which encoded a series of Pagefile operations. This input file could be generated by an actual run of ADMS or generated automatically to simulate a work load. The driver also included the instrumentation for timing its operation. In order to limit the effect of calling the UNIX timer, only large blocks of work were timed.

In addition to the workload file, the driver program accepts several parameters which control the specific experiments. These parameters define the character of the workload, whether the Pagefiles should be local or remote, and the level of concurrency for remote Pagefiles. Several parameters also govern the output of the experiment data. This allows the data from multiple experiments to be organized and cataloged and the proper graphs to be generated automatically. After testing all of these components the actual prototyping experiments were run.

5 EXPERIMENTS

The initial performance experiments include: an evaluation of concurrency; a test of total throughput; and a comparison of local and remote Pagefile performance. The concurrency test addresses the first question about achieving network speeds by spreading I/O across multiple servers. Total throughput is tested by varying the page size and checking the volume of data which can be read and written by the system. Finally the comparison between local and remote processing is made in order to establish the basic hypothesis. The other experiments needed to answer the questions about scalability and the locking and caching issues are not included in this first round of prototyping.

The configuration of the prototype for these experiments includes a single client running on a SPARC 10 and six disk servers running on a variety SPARC workstations. Two of the servers are SPARC IPXs, two are SPARC 2s, one is a SPARC 5 and the last server is second SPARC 10. The work load for the base performance experiments included 3000 I/O operations to a random pages in a single 16Mb file. Each point on each graph is the average of 7 to 9 execution runs using this workload.

5.1 Concurrency Experiments

The initial hypothesis was that a cluster of disks, attached by a network, could deliver pages at near network speeds. In order to achieve this, a sequence of page requests is made asynchronously. These requests activate the disks in the cluster concurrently. Each disk then returns the page (for read operations) or the confirmation (write operations) to the client. At some point the client must wait for a confirmation that the oldest of the requests has completed. The number of requests which are issued before this confirmation is the concurrency level. This is the number of requests which are in process at any given time. This parallelism is similar to instruction pipelining in modern central processors where, at a given time, different phases of several instructions are in the pipeline.

The first experiment is to quantify the effect of this concurrency and answer questions number 1 and 3. At a concurrency level of 1, pages are accessed synchronously. Each request is completed before any other request is issued. As the level of concurrency increases, more disks in the cluster are likely to be busy, and the number of pages returned to the client per second increases. At some point the disk cluster is saturated with requests and no increase in throughput can be realized. The first experiment quantifies the relationship between network latency and average page latency as the level of concurrency is increased.

Figure 5 shows this relationship for page sizes of 1k and 8k. The lines on both graphs show the number of milli-seconds per page which has been normalized by the total number of pages accessed. Although the latency of a single page is greater than would be experienced on a local disk, the average latency per page for multiple page requests is lower. The level of concurrency is the independent variable.

The lower line on each graph is the network latency alone. This is the overhead incurred by using the network. This is measured by re-reading pages in the remote cluster which are already cached from the remote disk. This represents a cache hit on the remote disk and, therefore, latency is the network overhead alone.

The higher line on each graph in figure 5 includes the network overhead and the overhead of disk access. This is measured by writing pages to a file which was opened using the `O_SYNC` option. This option insures that the data is written to the disk before control is returned to the calling program. It, therefore, represents the latency of a disk cache miss on the remote cluster.

The relationships quantify the effect of concurrency on throughput. For both 1k and 8k pages

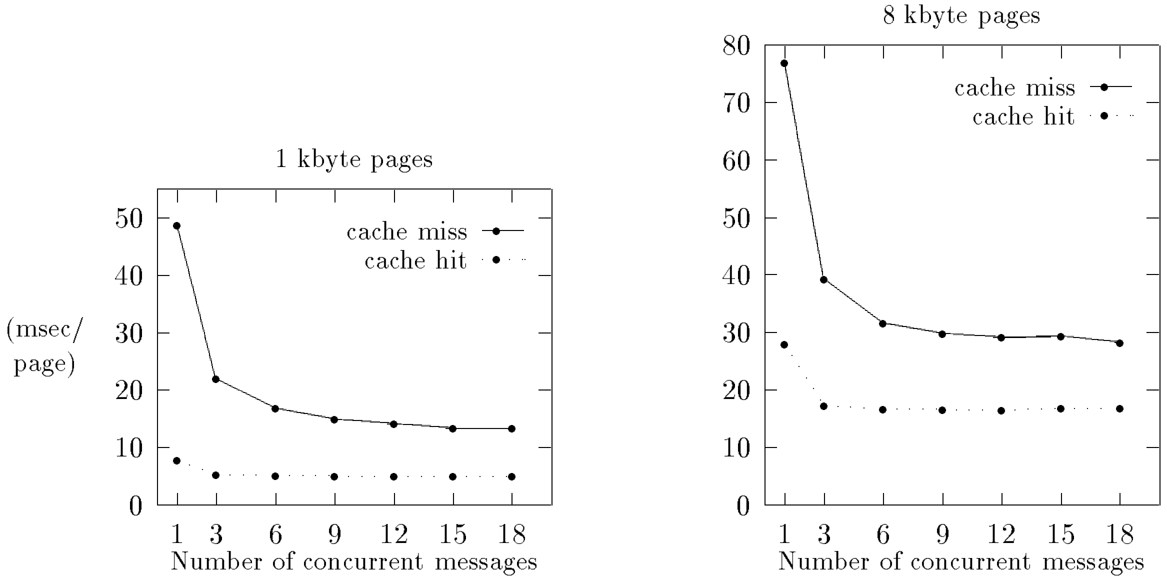

Figure 5: Average number of milli-seconds per page for various levels of concurrency.

the cache miss curves flatten out when there are about 12 request messages at a time in the system. This is intuitively consistent. If the messages happen to be evenly distributed over the 6 servers then the first 6 messages are needed to activate all servers. As the number of messages increases to 12, there is the potential for a message to be queued at each server while that server is accessing the disk. Beyond this, the number of messages in the network and the random nature of the requested pages prevents any dramatic improvement. This intuition needs to be confirmed with an additional experiment on a varying number of servers.

In addition to quantifying the initial hypothesis this experiment was designed to check the overhead of the prototype itself. The experiment provides numbers for the latency of the network and the latency of the disk access. The network overhead includes the system calls and context switches required to send a request message from one application program to another and return a 1k or 8k page. Other experiments have put the round trip network latency of 1 kb, in a TCP/IP network, between 4ms and 6ms. When the concurrency level is 1, the average time required for this round trip in our prototype is just over 8ms for 1k pages. Since our prototyping goal is to understand the effect of concurrency and not to minimize network overhead this is within an acceptable range.

Our own experiments with these disk servers put disk latency between 30ms and 50ms. These numbers are based on random writes to a 16mb file opened with the `O_SYNC` option. Figure 5 shows that the prototype is writing disk pages with a latency of 40ms to 50ms for 1k and 8k pages. This number is the difference between the two curves when the concurrency level is 1. These too are within an acceptable range when we are quantifying the effect of concurrency. The fact that the prototype itself is not introducing an large degree of overhead is one of the significant results of this first experiment.

5.2 Throughput Experiments

The second experiment uncovers some optimal parameters for the target system and answers question number 4. The overall throughput in bytes per second will depend, mostly, on the page size. This experiment quantifies the volume of data we can expect from this network and provides some guidelines for optimal use of the target system.

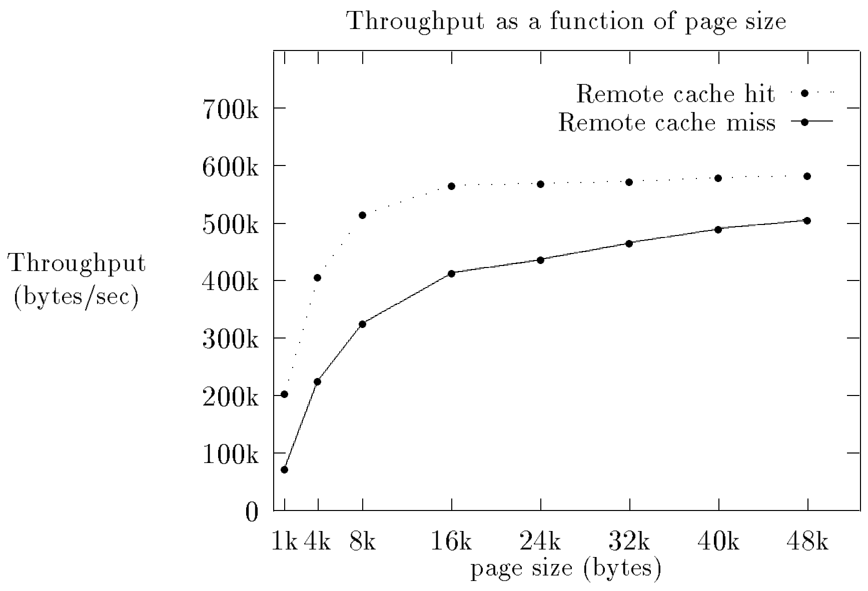

Figure 6: Throughput in bytes per second.

Figure 6 shows the effect of page size on throughput. Using the results from the first experiment, the concurrency level of this experiment was set to 12. The top line shows the throughput if all pages are found in the remote cache. This relationship shows the upper bound of throughput for this network and prototype. The lower line is the resulting throughput for no cache hits. Again, the cache miss was modelled using write operations on files which were opened with the `O_SYNC` option.

The throughput with cache hits quickly reaches the maximum of just over 500k per second. This is a reasonable maximum for a TCP/IP network based on 10Mbit Ethernet. The throughput, with no cache hits, increases dramatically up to page sizes of 16k and then tapers off. We conclude from this experiment that 500k bytes per second is the maximum we can expect from this network. In addition, the target system, of a similar configuration, may not be able to reach this maximum unless large page sizes can be used.

5.3 Local disk vs. Remote disk access

The third experiment compares the throughput of the remote cluster to the throughput for local access. This experiment answers part of question number 1 but it also quantifies the effect of disk page caching. The throughput using a local disk is highly dependent on the success of the disk caching provided by the operating system or, in the case of a DBMS, page buffering. The independent variable in this experiment is the percent of cache hits, remotely or locally. Accessing a page which is found in the remote cache still incurs the overhead of the network. Accessing a page found in the local cache incurs only the cost of a memory copy. This experiment also uses a concurrency level of 12.

Figure 7 compares local and remote access for 1k and 8k page sizes. The throughput of a local Pagefile drops off dramatically as the number of cache hits drops. This is a testament to the success of disk caching by operating systems. The effect of cache misses on remote processing is much more muted because the misses are incurred in parallel. This result increases our understanding of the performance parameters, and, for 8k page sizes, the experiment confirms the maximum of 500k byte/sec which was found in the previous experiment.

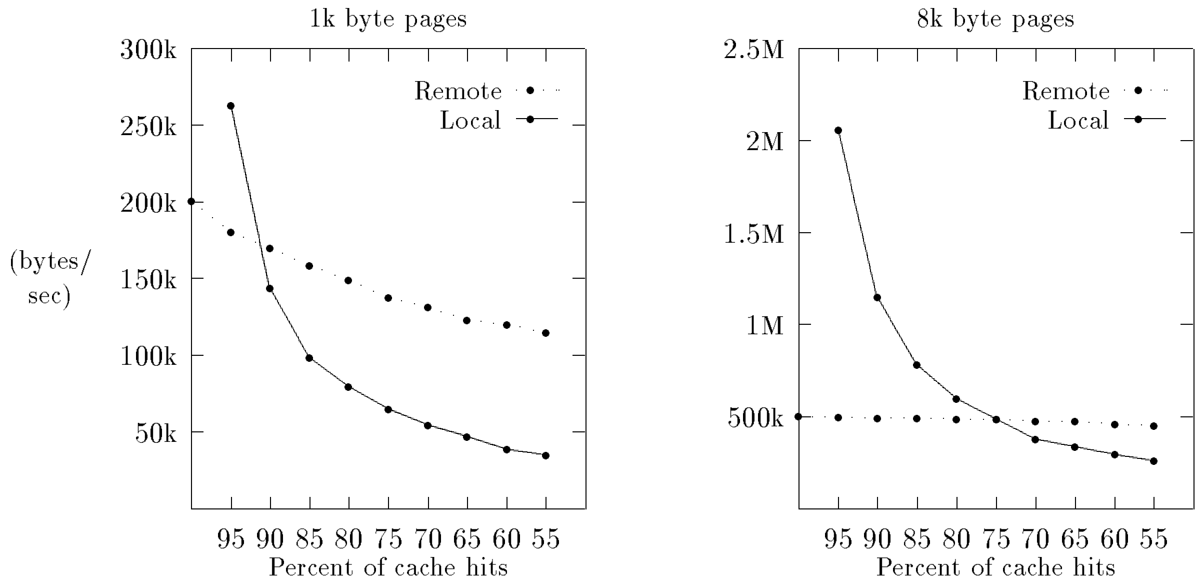

Figure 7: Throughput of local and remote files across for various levels of disk caching with a concurrency level of 12.

The significance of this experiment is that the in certain situation the remote cluster can outperform local disk access. When files are large and access is random, cache misses will be high. In these circumstances the throughput of the local disk will suffer and a the target system could be of great value. Secondly, the trend of throughput for remote files provide performance a expectation for target system across a wider range of applications.

5.4 Results of initial experiments

These experiments provide a basis for the initial evaluation of the a target system by answering some of the initial performance questions. They help build confidence in our understanding of how responsive the target system will be to an increase in concurrency, page size and the number of cache misses. By demonstrating that the prototype is operating within acceptable performance bounds we can reasonably expect the target system to operate in a similar manner. These results therefore qualify the situations in which a parallel disk cluster could provide improvement and quantify the performance we might expect from such a system.

The first experiment confirms our hope that concurrency will reduce average latency. The steep reduction in overall latency as the concurrency level moves from 1 to 6 is reflected in both graphs of figure 5. It is also interesting to note that the increase in overall page latency between 1k pages and 8k pages is mainly due to the network overhead. The disk latency, measured as the distance between the lines on each graph, is very similar for the 1k and 8k page sizes. The network latency, however, accounts for 35% of the overall latency of 1k pages but 58% of the 8k page latency.

The second experiment quantifies the network bottleneck while providing a trend in throughput for the increasing page sizes. Many application may not be able to make use of large page sizes and a distributed disk system would not prove to be a performance advantage. Once again, this experiment demonstrates and quantifies how dependent the target system will be on the performance of the interconnection network.

The final experiment illustrates the effect of local caching. The target system will not provide a performance improvement over local data in small files which will can stay in a large disk cache. For large files, however, which are not readily cached, spreading out the disk access can provide an improvement on overall access time. In addition, the dampened effect of increased cache misses will give more consistent performance.

Overall, the prototype demonstrates the environment in which the target system can provide a performance improvement. Applications which do not operate on large files or are generally unable to request multiple pages will not gain from a distributed disk cluster. The prototype also indicates how much the overall performance of the target system will depend on the performance of the network. This network dependency has a valuable corollary.

Figure 5 shows that the average page latency is mostly due to the network overhead and that this percentage will increase with page size. However, the speed of networks is improving faster than the speed of disks, suggesting that the performance from a remote disk cluster has more potential for improvement. For 8k pages, figure 5 shows that 58% of the overall latency is due to network overhead. This is an opportunity for an overall performance improvement as network performance increases. In figure 7, the local throughput will improve only as disk speeds increase, but the remote performance will improve with faster networks and with faster disks. The effect of improved network speeds on figure 6 is harder to gauge. Little effect would likely be seen at smaller page sizes but the maximum throughput would not be reach at page sizes of 16k and 32k.

Some of the other experiments that were done are not included here. While, to a lesser degree, they added to our understanding of the target system they do not add any more to the understanding of prototyping.

6 CONCLUSION

The goal of the tailored prototyping process is to focus effort on those issues critical to minimizing the risk that the larger system may fail. By spending time identifying the critical questions, and by concentrating the prototyping activity on answering those questions, the time and effort spent prototyping is minimized and the quality of the result is improved.

This paper has described the objectives of tailored prototyping, and has both motivated and illustrated the approach using details of a pilot project we ran specifically to assess our emerging understanding of the prototyping issues. Through this pilot, along with our application of tailored process to other applications at this site, we have refined the definition of tailored prototyping, while at the same time improving the corresponding applications. The pilot described here helped the developers of ADMS to understand the potential performance and use of and API for the target system. We expanded the emphasis on the cost and benefit of prototyping, and refined the initial step of identifying the risk associated with the design or implementation decisions. We were also able to improve our interconnection system by applying it to a new application domain.

The performance results for this pilot qualified the type of environment in which the target system could provide high performance: large files with access patterns which prohibited successful disk caching. The results also showed how dependent the system is on overall network performance. These results confirmed some of the initial expectations while restricting the environment in which we could expect to find high performance.

These results were more convincing to the developers because the prototype was constructive. The fact the that the prototype was a running system which passed pages across the network also assisted in answering some of the original questions. Other design issues such as locking and buffering could be discussed with substantiative result by referencing the prototype. In addition, the fact that the prototype could be used to deliver pages focused attention on the current methods and patterns of file access. Finally, the questions about the overhead and mechanics of page administration were answered.

The 'output' from our tailored process applied to this pilot, was the above information, along with the Pagefile artifact itself – and no more. In general, the nature and extent of deliverables will be dictated by the choice of risk hypotheses; we feel strongly that a rigorous traceability check is important when deciding what to carry away. Pressure from management to utilized intermediate prototyping apparatus, whether or not it relates to the conclusions, should be resisted as poor engineering. Hence, whether or not tailored prototyping can be been considered to be either "evolutionary" or "throw away" depends strictly upon the experiment.

In our pilot, the information is prime, but the code implementing the Pagefile system itself would also be delivered along with the information. Since the results were predicated upon our implementation choices within the system, to deliver only the information would risk having a later developer make alternate implementation decisions which might invalidate the experimental results. The software artifact, therefore, provides the necessary traceability back to the prototyping results. On the other hand, other parts of our apparatus, such as the test harnesses or front ends, could safely be discarded, as not being of relevance to later utilization of the performance strategy.

In retrospect, several aspects of the prototype were unnecessary. The prototype was built with a comprehensive error handling and a complete interface to the file system, including calls to `unlink` and `status`. By not following the tailored prototyping methodology in this regard the prototype took longer to build because it was over engineered. There did not exist questions about the ability of the system to recover from simple errors and therefore this feature should not have been in the prototype.

Our research in this area continues, as we seek larger scale projects for study. In addition, it is clear that we may now turn to consideration of what types of environmental and tool assistance might further improve the process. Unlike proposals from many other previous prototyping approaches, we feel that focus on decision support systems, collaborative problem solving tools, and configuration management mechanisms are likely to be the keys to greater productivity in prototyping.

The authors would like to acknowledge the help of Dr. Adam Porter, Larry Herman, and Bill Falkenberg. They all offered insight into the role of prototyping and its relation a target system.

References

[CPP94] C. Chen, A. Porter, and J. Purtilo. "Tool support for tailored software prototyping". In *Third Symposium on Assessment of Quality Software Development Tools*, June 1994.

[FHK93] C. Falkenberg, P. Hagger, and S. Kelley. "A server of distributed disk pages using a configurable software bus". Technical Report UMIACS-TR-93-47, Institute for Advanced Computer Studies at the University of Maryland, College Park, MD, June 1993.

[LSHZ93] H. Lichter, M. Schneider-Hufschmidt, and H. Züllighoven. "Prototyping in industrial software projects –Bridging the gap between theory and practice–". In *IEEE Proceedings of the International Conference on Software Engineering*, May 1993.

[PLC91] J. Purtilo, A Larson, and J Clark. "A methodology for prototyping-in-the-large". In *IEEE Proceedings of the International Conference on Software Engineering*, May 1991.

[Pur94] J. Purtilo. "The POLYLITH software bus". *ACM Transaction on Programming Languages*, 16(1):151–174, January 1994.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. "ADMS: A testbed for incremental access methods". *IEEE Transactions on Knowledge and Data Engineering*, 5(5):762–774, October 1993.

[Rou92] N. Roussopoulos. The tower of pizzas. Internal Memo, 1992.

[WK92] D. Wood and K. Kang. "A classification and bibliography of software prototyping". Technical Report CMU/SEI-92-TR-13, Software Engineering Institute, 1992.

James Purtilo is an Associate Professor of Computer Science at the University of Maryland, where he works in the area of languages and software engineering, with specialty in software interconnection and integration. He is a Senior Member of the IEEE, a voting member on several national software standards committees, and program chair for the next International Workshop on Configurable Distributed Systems.

Charles Falkenberg received a B.S. degree in computer science in 1992 from the University of Maryland and is currently working toward a Ph.D. degree in computer science at the University of Maryland. Before returning to school he worked as an independent systems analyst on large, transaction oriented, database applications. His current research includes scientific database management systems, and parallel and distributed systems.