Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization

by

Eric Allen Brewer

S.M., Massachusetts Institute of Technology (1992)
B.S, University of California at Berkeley (1989)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the
Massachusetts Institute of Technology
August 1994

Signature of Author ______________________
Department of Electrical Engineering and Computer Science
August 22, 1994

Certified by [illegible]
Professor William E. Weihl
Associate Professor of Electrical Engineering and Computer Science

Accepted by [illegible]
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Abstract

Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization

Eric Allen Brewer

Submitted to the Department of Electrical Engineering and Computer Science on August 22, 1994 in partial fulfillment of the requirements for the degree of Doctor of Philosophy under the supervision of Professor William E.Weihl.

Although there is some amount of portability across today's supercomputers, current systems cannot adapt to the wide variance in basic costs, such as communication overhead, bandwidth and synchronization. Such costs vary by orders of magnitude from platforms like the Alewife multiprocessor to networks of workstations connected via an ATM network. The huge range of costs implies that for many applications, no single algorithm or data layout is optimal across all platforms. The goal of this work is to provide high-level scientific libraries that provide portability with near optimal performance across the full range of scalable parallel computers.

Towards this end, we have built a prototype high-level library "compiler" that automatically selects and optimizes the best implementation for a library among a predefined set of parameterized implementations. The selection and optimization are based on simple models that are statistically fitted to profiling data for the target platform. These models encapsulate platform performance in a compact form, and are thus useful by themselves. The library designer provides the implementations and the structure of the models, but not the coefficients. Model calibration is automated and occurs only when the environment changes (such as for a new platform).

We look at applications on four platforms with varying costs: the CM-5 and three simulated platforms. We use PROTEUS to simulate Alewife, Paragon, and a network of workstations connected via an ATM network. For a PDE application with more than 40,000 runs, the model-based selection correctly picks the best data layout more than 99% of the time on each platform. For a parallel sorting library, it achieves similar results, correctly selecting among sample sort and several versions of radix sort more than 99% of the time on all platforms. When it picks a suboptimal choice, the average penalty for the error is only about 2%. The benefit of the correct choice is often a factor of two, and in general can be an order of magnitude.

The system can also determine the optimal value for implementation parameters, even though these values depend on the platform. In particular, for the stencil library, we show that the models can predict the optimal number of extra gridpoints to allocate to boundary processors, which eliminates the load imbalance due to their reduced communication. For radix sort, the optimizer reliably determines the best radix for the target platform and workload.

Because the instantiation of the libraries is completely automatic, end users get portability with near optimal performance on each platform: that is, they get the best implementation with the best parameter settings for their target platform and workload. By automatically capturing the performance of the underlying system, we ensure that the selection and optimization decisions are robust across platforms and over time.

Finally, we also present a high-performance communication layer, called Strata, that forms the implementation base for the libraries. Strata exploits several novel techniques that increase the performance and predictability of communication: these techniques achieve the full bandwidth of the CM-5 and can improve application performance by up to a factor of four. Strata also supports split-phase synchronization operations and provides substantial support for debugging.

Acknowledgments

Although the ideas in this work are my own, I clearly owe much to those around me for providing a challenging environment that remained supportive, and for providing a diversity of ideas and opinions that always kept me honest and regularly enlightened me. As with all groups, the nature of the environment follows from the nature of the leader, in this case Bill Weihl.

Bill leads by example: his real impact on me (and other students) comes from his belief in having a "life" outside of computer science, his support for collaboration over credit, his pragmatic approach towards "doing the right thing", and above all, his integrity. He gave me the freedom to explore with a perfect mixture of support and criticism.

Greg Papadopoulos and Butler Lampson were the other two committee members for this dissertation. They provided guidance and feedback on this work, and tolerated the problems of early drafts. It is hard to understate my respect for these two.

Bill Dally has been a guiding influence for my entire graduate career. I particularly enjoy our interactions because it seems like we always learn something new for a relatively small investment of time. There are few people willing to aim as high and long term as Bill; I hope I become one of them.

Charles Leiserson and I got off to a rough start (in my opinion) due to aspects of my area exam to which he correctly objected. I accepted his criticisms and improved my research methodology directly because of him. But more important to me, he accepted this growth as part of the process, never held my flaws against me, and in the end successfully helped me into a faculty position at Berkeley, apparently satisfied that I had improved as a researcher. All MIT grad students serious about faculty positions should talk to Charles.

Tom Leighton and I would have gotten off to a rough start, except he didn't realize who I was. Several years ago, I threw a softball from centerfield to home plate that was literally on the nose: it broke the nose of Tom's secretary, who was playing catcher. By the time Tom made the connection, we were already friends and colleagues through my work on metabutterflies. Besides playing shortstop to my third base, Tom gave me a great deal on insight on the theory community (and it is a community) and on the presentation of computer theory.

Frans Kaashoek was quite supportive of this work and always provided a different perspective that was both refreshing and motivating. He also gave me a lot of great advice on job hunting, negotiation, and starting a research group, all of which he recently completed with success.

There are so many people at MIT that I already miss. From my stay on the fifth floor, I built friendships with Anthony Joseph, Sanjay Ghemawat, Wilson Hsieh, and Carl Waldspurger. I have a great deal of respect for all of them. Debby Wallach, Pearl Tsai, Ulana Legedza, Kavita Bala, Dawson Engler and Kevin Lew have added a great deal to the group; they are well equipped to lead it as my "generation" graduates.

The CVA group, led by Bill Dally, also contains a great bunch of people. I've had some great times with Peter and Julia Nuth, Steve Keckler, Stuart Fiske, Rich Lethin, and Mike Noakes. Lisa and I will miss the CVA summer picnics at Bill's house.

In addition to many technical discussions, David Chaiken, Beng-Hong Lim and I commiserated about the machinations of job hunting and negotiation. Kirk Johnson and John Kubiatowicz influenced me both with their technical knowledge and their unyielding commitment to "do the right thing." Fred Chong and I have worked together several times, and it has always been a good match. We tend to have different biases, so it is always fruitful to bounce ideas off of him. I look forward to our continued collaboration.

Moving to the second floor (along with the rest of Bill Weihl's group) was quite beneficial to me. In addition to my interaction with Charles, I met and collaborated with Bobby Blumofe and Bradley Kuszmaul. All three of us have different agendas and backgrounds, so our conversations always seems to lead to something interesting, especially in those rare instances where we come to a conclusion! Bradley's insight on the CM-5 was critical to the success of our work together. Bobby and I seem to take turns educating each other: I give him a better systems perspective and he gives me a better theory perspective. If nothing else, we at least guide each other to the "good" literature.

My office mates, Shail Aditya and Gowri Rao, were always pleasant and supportive. In particular, they tolerated both my mess (especially at the end) and the large volume of traffic into my office. I hope whatever they got from me was worth it.

My parents, Ward and Marilyn, and my sister, Marilee, also helped me achieve this degree. My dad always took the long-term view and was influential in my selection of MIT over Berkeley and Stanford, which would have been easy choices for a native Californian, but would have been worse starting points for a faculty position at Berkeley, which has been a goal at some level for seven years. My mom contributed most by her undying belief in positive thinking and making things go your way. Marilee, a graduate student in materials science, always provided a useful perspective on graduate student life, and implicitly reminded me why I was here.

Although I had many technical influences during my graduate career, the most important person in the path to graduating relatively quickly was my fiancé Lisa. She both inspired me to achieve and supported me when it was tough. It has been a long path back to Berkeley, where we met as undergrads, but we made it together and that made it worthwhile.

Table of Contents

List of Figures

List of Tables

Introduction

Although the hardware for multiprocessors improved greatly over the past decade, the software and development environment has made relatively little progress. Each vendor provides its own collection of languages, libraries, and tools: although similar in name or spirit, these systems are never quite compatible.

There is, however, hope on the horizon. High-Performance Fortran (HPF) seeks to standardize a version of Fortran for multiprocessors, and Gnu C and C++ are becoming de facto standards as well. There are also some shared third-party libraries, such as LAPACK for linear algebra. There is reason to believe that some level of portability for C and Fortran is fast approaching.

However, these plans lead to only the weakest form of portability: the ported application will run correctly, but may not run *well*. In fact, it is very likely that the application will perform poorly on the new platform. There are two main reasons for this expectation: first, different platforms often require different algorithms or data layouts to achieve good performance. Second, low-level languages such as C and Fortran force the user to encode such decisions in the source code. Together, these two properties result in applications that are tuned for a single architecture. Moving to a new platform requires extensive changes to the source code, which requires a substantial amount of time and effort, and often introduces subtle new bugs.

These properties are fundamental rather than temporary idiosyncrasies of the vendors. The key reason behind the need for different algorithms and data layouts is the tremendous variance in the relative costs of communication and computation across modern supercomputers. Hardware support for communication and synchronization varies greatly across current supercomputers, especially when considering vector supercomputers, massively parallel processors, and networks of workstations. The operating-system support also varies substantially, with some allowing user-level message passing and others restricting communication to the kernel, which costs substantially more. For example, networks of workstations currently promote a small number of large messages, since they have high start-up overhead both in hardware and software, while machines like the CM-5 support very small messages. These kinds of differences lead to different conclusions about the best algorithm or data layout for a given platform.

The encoding of low-level decisions in the source code is fundamental to C and Fortran, but can be avoided by moving such decisions into libraries, or by using higher-level languages. In this work, we achieve portability with high-performance by moving the key performance decisions into libraries, where they can be changed without affecting the source code.

However, this is only part of the answer. Although libraries allows us to change our decisions easily, we still must determine the appropriate outcome for each decision on the new platform. The real contribution of this work is a novel technology that can make all of these decisions automatically, which results in truly portable applications: we get the *correctness* from the portability of the underlying language, and the *performance* from the ability to change and re-tune algorithms automatically.

There are many reasons why it is difficult to make these decisions automatically, but two stand out as truly fundamental. First, the trade-offs that determine the best algorithm or parameter setting depend on the platform, the libraries, and the operating system. Even if you knew the "crossover points" for one platform, those values would not apply for another machine. Thus, any system that makes such decisions automatically and in a portable manner must be aware of the underlying costs of the specific target platform.

Second, there is no general mechanism or framework by which the decision maker can obtain cost information. Currently, costs for a particular platform, if used at all, are embedded into the compiler by hand. Besides being non-portable, this mechanism is quite error prone: the costs may or may not be accurate and are likely to become obsolete over time. Ideally, we would like a framework in which the underlying costs are always accurate, even as we move to new platforms.

Both of these reasons apply to humans as well as to compilers. There are a few systems in which some decisions can be easily changed by the programmer without affecting the (primary) source code or the correctness of the application. High-Performance Fortran, for example, provides directives by which the user can control data layout. The existence of these directives follows from the platform-dependence of the best layout. However, there is no support for making these decisions: users must guess the best layout based on their expectations of the platform. From our perspective, HPF solves only half of the problem: it provides the ability to update decisions, but no way to do so automatically or even reliably.

Thus, high-level decisions such as algorithm selection require an infrastructure that can evaluate the impact of each option. To ensure portability, this framework must be robust across architectures. To ensure accuracy over time, it must reflect the current costs of the platform. Given such an infrastructure, and the encapsulation of the decisions provided by libraries, we can, in theory, build a "compiler" that automatically selects the best algorithm (from a fixed set). Furthermore, we can optimize each of the algorithm's parameters for the target platform and workload. This thesis describes such a system based on an infrastructure provided by statistical models.

The resulting libraries, called *high-level libraries*, combine a set of algorithms that provide a unified interface with a selection mechanism that picks the best algorithm for the current platform and workload. The selector also uses the statistical models to determine the optimal value for the algorithm parameters. Portability comes from the ability to recalibrate the models for a new platform. Robustness over time comes from the ability to recalibrate the models whenever the environment changes, such as after an update to the operating system. Thus, the end users receive the best available performance: they always get the best algorithm with the best parameter settings, even though the specific choices change across platforms, across workloads, and over time. It is this "under-the-covers" selection and optimization that provides true portability, that is, portability with high performance.

1.1 System Overview

Figure 1-1 shows the relationship among the various components of this thesis. Each high-level library consists of several implementations, the support for parameter optimization and the support for algorithm selection. The latter two are shared among all high-level libraries, but it is useful to think of them as part of each library.

The models are based on profiling data from the target platform that is collected automatically. We use statistical regression to convert profiling data into a complete model that covers the entire range of the input space. The models predict the performance of each implementation for the given platform and input problem.

Both the parameter optimizer and the selector use the models to predict the performance of the various options. For example, the selector evaluates each implementation's models for the expected workload to obtain predicted execution times. The implementation with the smallest execution time is selected as the best algorithm. Similarly, the parameter optimizer uses the models to find the parameter setting that leads to the best performance.

The key to the success of this approach is the accuracy of the models: if the models are accurate the decisions will be correct and the end user will receive the best performance. To obtain accurate models we combine two techniques: profiling, to collect accurate performance information for particular executions, and statistical modeling, to convert a small number of samples into a multi-dimensional surface that predicts the execution time given a description of the input problem. The *auto-calibration toolkit* automates the data collection and modeling steps to produce models for each implementation.

Profiling provides several nice properties. First, we can treat each platform as a black box: profiling gives us an accurate sample of the system's behavior without any platform-specific information. This allows us to move to a new platform without changing the infrastructure; we only require that the run-time system executes on the new platform, since it supports all of the libraries and provides the timing mechanisms used for profiling. The second nice property of profiling is that it captures all facets of the system: the hardware, the operating system, and the C compiler. Difficult issues such as cache behavior are captured automatically. Finally, profiling is easily automated, since all we really need is some control structure to collect samples across the various dimensions of the problem space.

Statistical modeling allows us to convert our profiling data into something much more powerful: a complete prediction surface that covers the entire input range. The key property of statistical

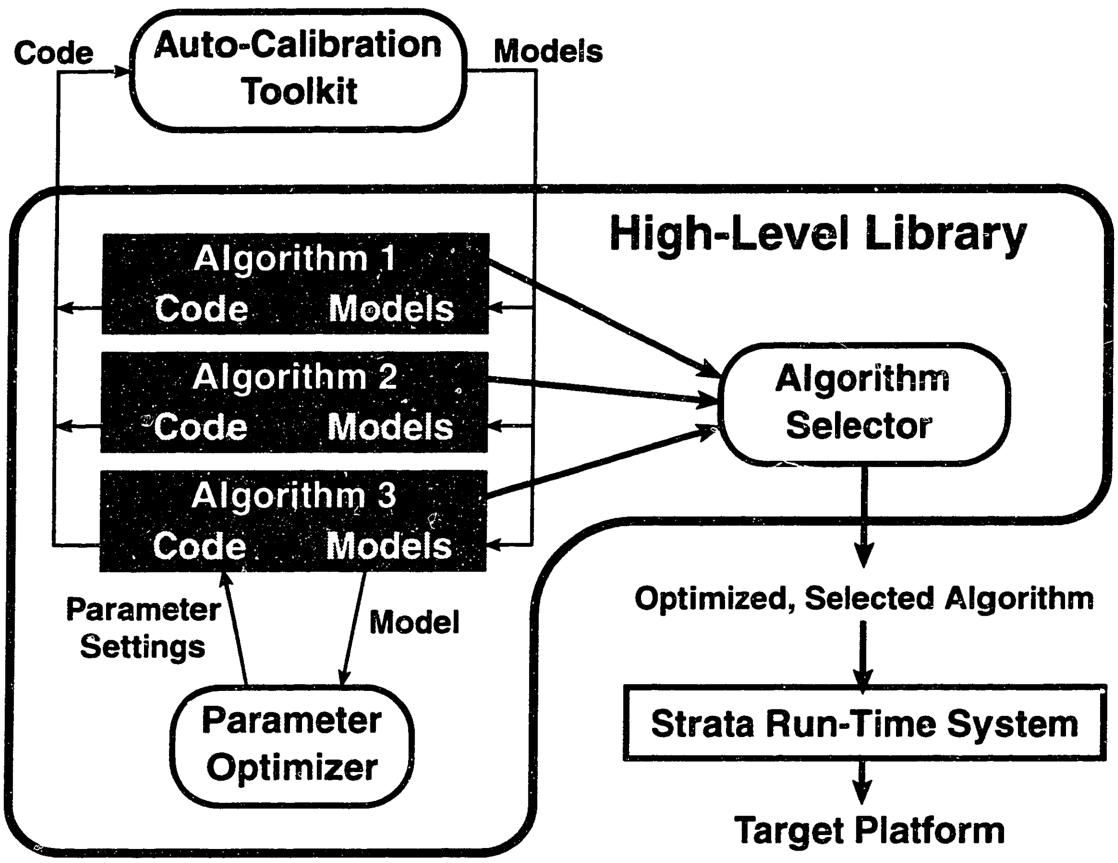


Figure 1-1: High-Level Library Overview

This diagram shows the components of a high-level library and the relationship of the various pieces of the thesis. A library contains a set of parameterized implementations, each with both code and models for the target platform. The auto-calibration toolkit generates the models by timing the code. These models form the basis for automatic selection and parameter optimization. After selecting and optimizing the best algorithm, the system compiles that implementation with the Strata run-time system to produce an executable for the target platform.

modeling in this context is that it can build accurate models with very little human intervention. Other than a list of terms to include in the model, there is no substantial human input in the model building process. The toolkit determines the relevant subset of the terms and computes the coefficients for those terms that lead to the best model. It then validates the model against independent samples and produces a summary metric of the accuracy of each model.

The designer of a high-level library must provide a list of terms on which the performance of the library depends, although he can include terms that end up being irrelevant. The list might include items such as the number of processors, the size of the input problem, or other basic metrics. The list often also includes functions of the basics. For example, a heap-sort algorithm would probably have an $n \log n$ term, where n is the size of the problem, since that term represents the asymptotic performance of the algorithm. This list of terms can be shared for all of the implementations and generally for all of the platforms as well, since the list can simply be the union of the relevant terms for all platforms and implementations. The list captures the designer's knowledge about the performance of the algorithm in a compact and abstract form. The designer only needs to know the overall factors that affect the performance without knowing the specific impact for the target platform. Since these factors are generally independent of the platform, the list is portable, even to future multiprocessors. Thus, the toolkit fits the same list to different platforms by throwing out irrelevant terms and adjusting the coefficients of the remaining terms.

Given the best algorithm and the optimal settings for its parameters, the selected implementation is then compiled and linked with the run-time system. This work also presents a novel run-time system, called Strata, that both supports the libraries and provides tools for the data collection required for model building. Strata provides several contributions that are useful beyond the scope of high-level libraries, including extremely fast communication, support for split-phase global operations, and support for development. Of particular importance are novel techniques that improve communication performance for complex communication patterns by up to a factor of four. Strata implicitly helps high-level libraries by providing a clean and predictable implementation layer, especially for complex global operations and communication patterns. The predictable primitives greatly increase the predictability of the library implementations, which leads to more accurate models and better selection and parameter optimization.

1.2 Programming Model

Strata provides a distributed-memory SPMD programming model. The SPMD model (single program, multiple data) structures applications as a sequence of phases, with all processors participating in the same phase at the same time. The processors work independently except for the notion of phases. For example, in a sorting application, the phases might alternate between exchanging data and sorting locally. The local sorting is completely independent for each node, but all nodes enter the exchange phase at the same time, usually via a barrier synchronization.

Alternative models include the SIMD model (single instruction, multiple data), in which all nodes execute the same instruction sequence in unison. This model also has phases in practice, but eliminates independence among nodes within a phase (compared to SPMD). At the other extreme is the MIMD model (multiple instruction, multiple data), in which each node is completely independent and there is no notion of phases.

Although the MIMD model is more flexible, the absence of global control makes it very difficult to make effective use of shared resources, particularly the network. In fact, we show in Chapter 7 that the global view provided by the SPMD model allows us to reach the CM-5's bandwidth limits even for complex communication patterns. Without this global view, performance can degrade by up to a factor of three, even if we exploit randomness to provide better load balancing.

The SPMD model has a related advantage over the MIMD model for this work. Because the phases are independent, we can compose models for each phase into a model for the entire application just by summation. In contrast, in the MIMD model the overlapping "phases" could interfere, which would greatly reduce the accuracy of model predictions. For example, if we have accurate models for local sorting and for data exchange, we may not be able to compose them meaningfully if half of the nodes are sorting and half are trying to exchange data. Some nodes might stall waiting for data, while those sorting would be slowed down by arriving messages. The uses of phases provides a global view of shared resources that not only improves performance, but also greatly improves predictability, since all nodes agree on how each resource should be used.

The SPMD model also has a limitation: it may be difficult to load balance the nodes within a phase. The MIMD model thus could have less idle time, since nodes can advance to the next phase without waiting for the slowest node. Of course, this only helps if the overall sequence is load balanced; otherwise, the fast nodes simply wait at the end instead of waiting a little bit in each phase. Techniques like work stealing that improve load balancing in the MIMD model also apply to the SPMD model within a phase, so for most applications there is no clear advantage for the MIMD model.

Another facet of the programming model is the assumption of a distributed-memory multiprocessor, by which we mean separate address spaces for each node. Distributed-memory machines are typically easier to build and harder to use. The most common alternative is the shared-memory multiprocessor, in which there is hardware support for a coherent globally shared address space. In fact, we investigate implementations that involve both classes, but we use distributed-memory machines as the default, since they are generally more difficult to use well and because the distributed-memory algorithms often run well on shared-memory machines, while shared-memory algorithms are generally unusable or ineffective on distributed-memory machines. A third alternative is a global address space without hardware support for coherence. For these machines, explicit message passing leads to the best performance, while the global address space acts to simplify some kinds of operations, particularly those involving shared data structures.

To summarize, this work assumes a distributed-memory SPMD model. The key benefits of this model are the performance and predictability that come from a global view of each phase. The independence of the phases leads to independent models and thus to easy composition. Finally, we have found that the existence of phases simplifies development, since there are never interactions among the phases: phases can be debugged completely independently.

1.3 Summary of Contributions

This thesis develops the technology required for portability with high performance. We build two high-level libraries and show how the system can successfully select and optimize the implementations for each one. The resulting libraries are easy to port and, more importantly, support portable applications by moving performance-critical decisions into the library, where they can be adjusted for the target platform and workload.

Although the primary contributions involve the techniques for algorithm selection and parameter optimization, there are several other important contributions: the techniques and automation of the auto-calibration toolkit, the Strata run-time system, and the novel techniques for high-level communication that improve performance and predictability.

1.3.1 Automatic Selection and Parameter Optimization

We implement two high-level libraries: one for iteratively solving partial differential equations, and one for parallel sorting. Each has multiple implementations and models that allow automatic selection of the best implementation. The selector picks the best implementation more than 99% of the time on all of the platforms.

In the few cases in which a suboptimal implementation was selected, that implementation was nearly as good as the best choice: only a few percent slower on average for the stencils and nearly identical for sorting. The benefit of picking the right implementation is often very significant: averaging 8-90% for stencils and often more than a factor of two for sorting.

Automatic selection allows tremendous simplification of the implementations. This follows from the designer's ability to ignore painful parts of the input range. By adding preconditions into the models, we ensure that the selector never picks inappropriate implementations. Thus, an immediate consequence of automatic selection is the ability to combine several simple algorithms that only implement part of the input range into a hybrid algorithm that covers the entire range.

An extension of algorithm selection is parameter optimization, in which we develop a family of parameterized implementations. The parameter can be set optimally and automatically: we found no errors in the prediction of the optimal parameter value. Model-based parameter optimization provides a form of adaptive algorithm that remains portable. In fact, one way to view this technique is as a method to turn an algorithm with parameters that are difficult to set well into adaptive algorithms that compute the best value. As with algorithm selection, this brings a new level of performance to portable applications: the key performance parameters are set optimally and automatically as the environment changes.

1.3.2 The Auto-Calibration Toolkit

We also develop a toolkit that automates much of the model-building process. The model specifications are simple and quite short, usually about one line per model. Furthermore, the toolkit removes irrelevant basis functions automatically, which allows the designer to add any terms that he believes *might* be relevant.

The code for sample collection is generated and executed automatically, and the samples are robust to measurement errors. The toolkit tests each produced model against independent samples to ensure that the model has good predictive power. Finally, the toolkit produces C and perl versions of the models that can be embedded within a larger system, such as the tools for algorithm selection and parameter optimization.

1.3.3 The Strata Run-Time System

We also present the Strata run-time system, which provides fast communication, split-phase global operations, and support for development. Strata provides active messages that are extremely fast: we obtained speedups of 1.5 to 2 times for a realistic sparse-matrix application compared to other active-message layers for the CM-5. Strata's active-message performance comes from efficient implementation, effective use of both sides of the network, and flexible control over polling.

The block-transfer functions achieve the optimal transfer rate on the CM-5, reaching the upper bound set by the active-message overhead. The key to sustaining this bandwidth is a novel technique called *bandwidth matching* that acts as a static form of flow control with essentially zero overhead. Bandwidth matching increases performance by about 25% and reduces the standard deviation for the performance of complex communication patterns by a factor of fifty.

Strata provides timing and statistical operations that support the data collection required for automatic model generation. It also provides support for development that includes printing from handlers, atomic logging, and integrated support for graphics that provide insight into the global behavior of a program.

All of the Strata primitives have accurate performance models that were generated in whole or in part by the auto-calibration toolkit. These models lead to better use of Strata as well as to improvements in the Strata implementations. Strata is extremely predictable, which leads to more predictable applications and thus improves the power of statistical modelling for algorithm selection and parameter optimization.

1.3.4 High-Level Communication

We developed three mechanisms that lead to successful high-level communication such as the transpose operation: the use of barriers, the use of *packet interleaving*, and the use of *bandwidth matching*. We show that by building complex communication patterns out of a sequence of permutations separated by barriers, we can improve performance by up to 390%.

With packet interleaving, we interleave the data from multiple block transfers. This reduces the bandwidth required between any single sender-receiver pair and also eliminates head-of-line blocking. We use this technique to develop a novel asynchronous block-transfer module that can improve communication performance by more than a factor of two.

We show how Strata's bandwidth matching improves the performance of high-level communication patterns and provide rules of thumb that reveal how to implement complex communication patterns with maximum throughput. Finally, we discuss the implications of these mechanisms for communication coprocessors.

1.4 Roadmap

Chapter 2 provides the background in statistical modeling required for the rest of the dissertation. Chapter 3 covers the auto-calibration toolkit, which automates most of the model-generation process, including data collection, model fitting, and verification. In Chapter 4, we describe the use of statistical models for automatic algorithm selection, and in Chapter 5 we extend the techniques to cover parameterized families of implementations. We move down to the run-time system in the next two chapters, with Chapter 6 covering Strata and Chapter 7 describing the techniques and modules for high-level communication on top of Strata. Finally, Chapter 8 looks at extensions and future work and Chapter 9 presents our conclusions. The bibliography appears at the end, along with an appendix containing the Strata reference manual.

Statistical Modeling

This chapter provides the background in statistical modeling required for the rest of the dissertation. The notation and definitions are based on those of Hogg and Ledolter from their book *Applied Statistics for Engineers and Physical Scientists* [HL92]. Additional material came from *Numerical Recipes in C: The Art of Scientific Computing* by Press, Teukolsky, Vetterling, and Flannery [PTVF92], and from *The Art of Computer Systems Performance Analysis* by Jain [Jai91].

2.1 What is a model?

A model seeks to predict the value of a measurement based on the values of a set of input variables, called the *explanatory variables.* For example, one model might estimate the gas mileage for a car given its weight and horsepower as explanatory variables. The model is a function, f:

$$\text{Miles Per Gallon} \approx f(\text{weight, horsepower}) \qquad (1)$$

Throughout the text, we use y for outputs and x for explanatory variables. Figure 2-1 illustrates these definitions for the basic linear-regression model, and Table 2-1 summarizes the notation. First, the *measured* value for the i^{th} sample is denoted by y_i, $\hat{y}_i$ denotes the corresponding model prediction, and $\bar{y}$ denotes the average of the measurements. Similarly, x_i denotes the i^{th} input value for models with a single explanatory variable, and $x_{i,j}$ denotes i^{th} value of the j^{th} explanatory variable. The average over all samples for the single-variable case is denoted by $\bar{x}$.

Notation	Definition
y_i	Measured value for the i^{th} sample
$\hat{y}_i$	Predicted value for the i^{th} sample
$\bar{y}$	The average of the measured values
ε_i	The i^{th} residual; $\varepsilon_i \equiv y_i - \hat{y}_i$
σ	The standard deviation of the residuals, ε_i
$x_{i,j}$	Value of the j^{th} explanatory variable for the i^{th} sample
$\vec{x}_i$	The vector of the explanatory variables for the i^{th} sample
$\bar{x}$	The average value of the (single) explanatory variable
β_j	The coefficient for the j^{th} explanatory variable
$\vec{\beta}$	The vector of all coefficients, $\vec{\beta} \equiv \left[\beta_0\ \beta_1\ \ldots\ \beta_m\right]$

Table 2-1: Summary of Notation

Given that models are never perfect it is important to examine the prediction errors. Each estimate has a corresponding error value, which is called either the *error* or the *residual*. The error between the i^{th} measurement and its prediction is denoted by ε_i and is defined as:

$$\varepsilon_i \equiv y_i - \hat{y}_i . \tag{2}$$

It is important to separate the quality of a model in terms of its *predictive power* from the quality of a model in terms of its reflection of the underlying *cause and effect*. For example, a model that successfully predicts gas mileage only in terms of car weight and engine horsepower says nothing about how a car works or about why the predictive relationship even exists. This kind of distinction is used by tobacco companies to argue that the high correlation between smoking and lung cancer proves nothing about the danger of smoking. The distinction is important for this work because it is much simpler to establish a predictive model for a computer system than it is to build a model that accurately reflects the underlying mechanisms of the system, especially for models intended for a variety of platforms.

Finally, given a model it is critical to *validate* the model. A good model should be able to predict outputs for inputs that are similar but not identical to the ones used to build the model. For example, one model for gas mileage simply records the weight, horsepower and gas mileage for the input set, which produces a function that is just a map or a table lookup. This is a poor model, however, since it has no predictive power for cars outside of the original input set. For our purposes, validating a model means applying it to some new and independent data to confirm that the predictive power remains.

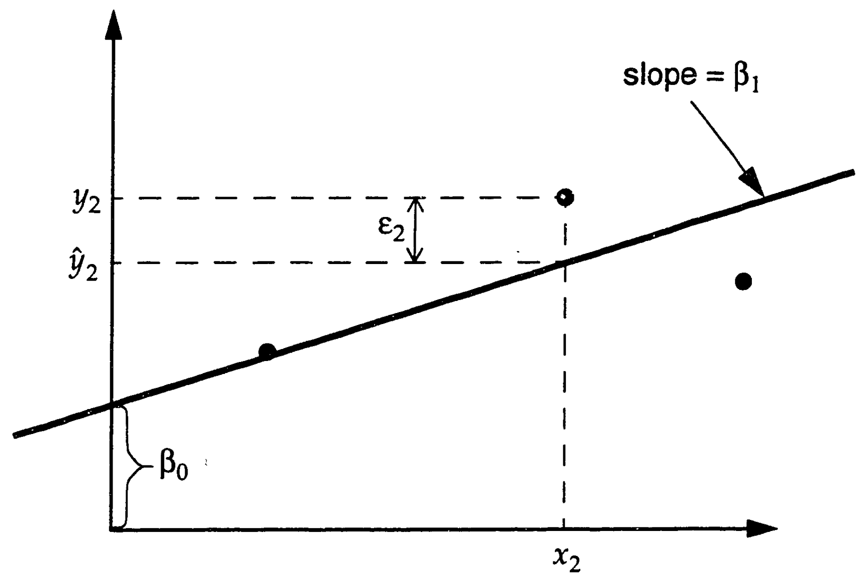


Figure 2-1: Components of the Simple Linear-Regression Model

2.2 Linear-Regression Models

This section looks at the simplest linear-regression model, which has only one explanatory variable. This model has the form:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \tag{3}$$

with the following assumptions, which are illustrated in Figure 2-1:

1. x_i is the i^{th} value of the explanatory variable. These values are usually predetermined inputs rather than observed measurements.
2. y_i is the measured response for the corresponding x_i.
3. β_0 and β_1 are the coefficients, or *parameters*, of the linear relationship, with β_0 as the intercept and β_1 as the slope.
4. The variables $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ are random variables with a normal distribution with a mean of zero and a variance of σ^2 (a standard deviation of σ). They are mutually independent and represent the errors in the model and the measurements.

Given these assumptions it follows that y_i comes from a normal distribution with an expected value of $\beta_0 + \beta_1 x_i$ and a variance of σ^2. Furthermore, all of the y_i are mutually independent, since they are simply constants plus independent error terms.

2.2.1 Parameter Estimation

Given that we have decided to model a data set with a linear model, the task becomes to find the best estimates for β_0 and β_1. Although there are many ways to define "best", the standard definition, known as the *least-squares estimates*, is the pair, $\vec{\beta} \equiv \begin{bmatrix} \beta_0 & \beta_1 \end{bmatrix}$, that minimizes the sum of the squares of the errors:

$$\sum_{i=1}^{n} (\varepsilon_i)^2 \equiv \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \equiv \sum_{i=1}^{n} [y_i - (\beta_0 + \beta_1 x_i)]^2 \tag{4}$$

To find the $\vec{\beta}$ that minimizes (4), we take the partial derivatives and set them equal to zero. The resulting set of equations form the *normal equations*; for the single-variable case they have the following solution:

$$\beta_1 = \frac{\sum_i (x_i - \bar{x})\, y_i}{\sum_i (x_i - \bar{x})^2} \tag{5}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \tag{6}$$

Thus, given a data set we can compute the β_0 and β_1 that produce the "best" line for that set using equations (5) and (6).

2.2.2 An Example

Given these tools we can build a model that predicts gas mileage based on the weight of a car. Table 2-2 provides the fuel consumption and weight for a variety of cars. The data is from 1981 and first appeared in a paper by Henderson and Velleman [HV81]. Note that fuel consumption is given in gallons per 100 miles rather than miles per gallon, since we expect the miles per gallon to be *inversely* proportional to the weight of the car. Using equations (5) and (6), we compute the coefficients as:

$$\beta_0 = -0.363 \qquad \beta_1 = 1.64 \tag{7}$$

Figure 2-2 shows a plot of the data with the fitted model and a plot of the residuals. From the first plot we confirm that using gallons per mile did lead to a linear model. If there were patterns in the residual plot, shown on the right, we might consider a more complicated model to capture that additional information. However, since the residuals appear to be random, we determine that our simple model is sufficient.

Car	Gallons per 100 Miles	Weight (1000 Pounds)
AMC Concord	5.5	3.4
Chevy Caprice	5.9	3.8
Ford Wagon	6.5	4.1
Chevy Chevette	3.3	2.2
Toyota Corona	3.6	2.6
Ford Mustang Ghia	4.6	2.9
Mazda GLC	2.9	2.0
AMC Sprint	3.6	2.7
VW Rabbit	3.1	1.9
Buick Century	4.9	3.4

Table 2-2: Gas-Mileage Data for Several Cars

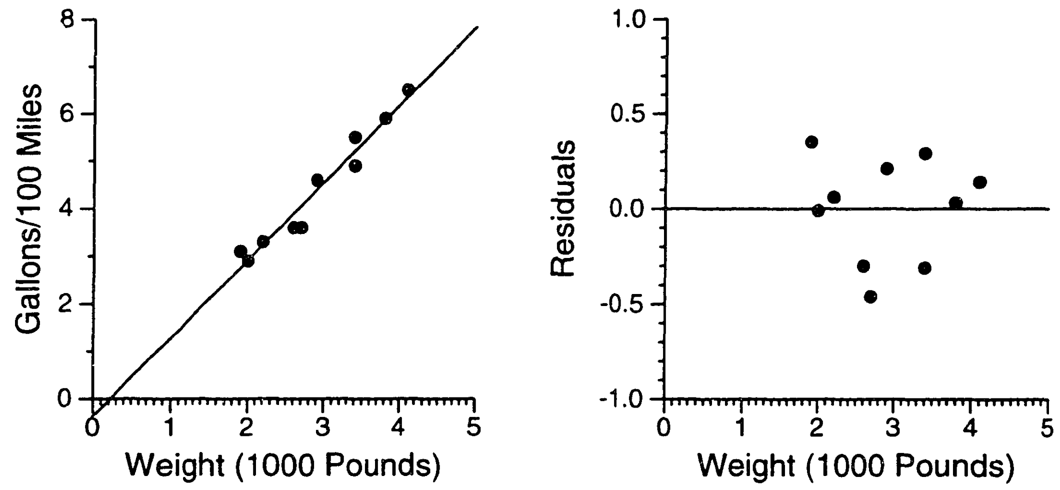


Figure 2-2: Mileage Plots

The left graph combines a scatter plot of the fuel consumption and weight data from Table 2-2 with a line for the linear-regression model. The graph on the left plots the residuals for this model against the explanatory variable. The data looks linear and the residual plot has the desired random spread of data.

2.2.3 Model Evaluation

Given the values of $\vec{\beta}$, how can we verify that we have a reasonable model? For humans, the best check is to simply look at a plot of the data and the fitted line: it is usually quite obvious whether or not the line "predicts" the data.

The plot of the residuals is similar, but more powerful for models with many explanatory variables. As we saw with the preceding example, if the line matches the data, then the residuals should be randomly placed around zero, since the errors are supposedly independent and come from a normal distribution with a mean of zero.

Although plots are great for people, they are useless for automatic evaluation of a model. The most common numerical model evaluation is the *coefficient of determination*, R^2, which summarizes how well the variance of the data matches the variance predicted by the model. R^2 ranges from zero to one, with good models having values close to one. For the one-variable case, we get:

$$R^2 \equiv \frac{\sum_i (\hat{y}_i - \bar{y})^2}{\sum_i (y_i - \bar{y})^2} \qquad 0 \le R^2 \le 1 \tag{8}$$

Although this allows us to evaluate a model numerically, it turns out to be insufficient for the kinds of models that arise in computer-systems modeling. Figure 2-3 shows four data sets with identical least-squares models and R^2 values, even though the models clearly differ in quality. Set (a) shows a typical data set for which the line is a decent model. In (b), the line is simply a poor approximation for a higher-order curve. The data in (c) is linear except for an outlier, which throws off the computed slope, while the data in (d) lacks sufficient variance along the x axis to merit a model in terms of x. A key point of these graphs is that the coefficient of determination is relatively poor as a numerical evaluation of the quality of a model.

2.2.4 Mean Relative Error

The models developed in this thesis tend be quite accurate and thus require a better measure of quality than that provided by R^2. The fundamental limitation of R^2 is that it measures only the degree to which the variance of the model estimates match the variance of the data. We would like an evaluation function that looks at the errors more directly.

Furthermore, not all errors are of equal importance: an error of 10 on a measurement of 1000 is minor compared to an error of 10 for a measurement of 10. Thus, we would like a metric that examines the relative error of the estimates rather than the absolute error. To meet these criteria, we propose a novel metric, the *mean relative error (MRE)*. First, we define the relative error of an estimate, r_i, as:

$$r_i \equiv 1 + \left|\frac{\varepsilon_i}{y_i}\right| \equiv 1 + \left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{9}$$

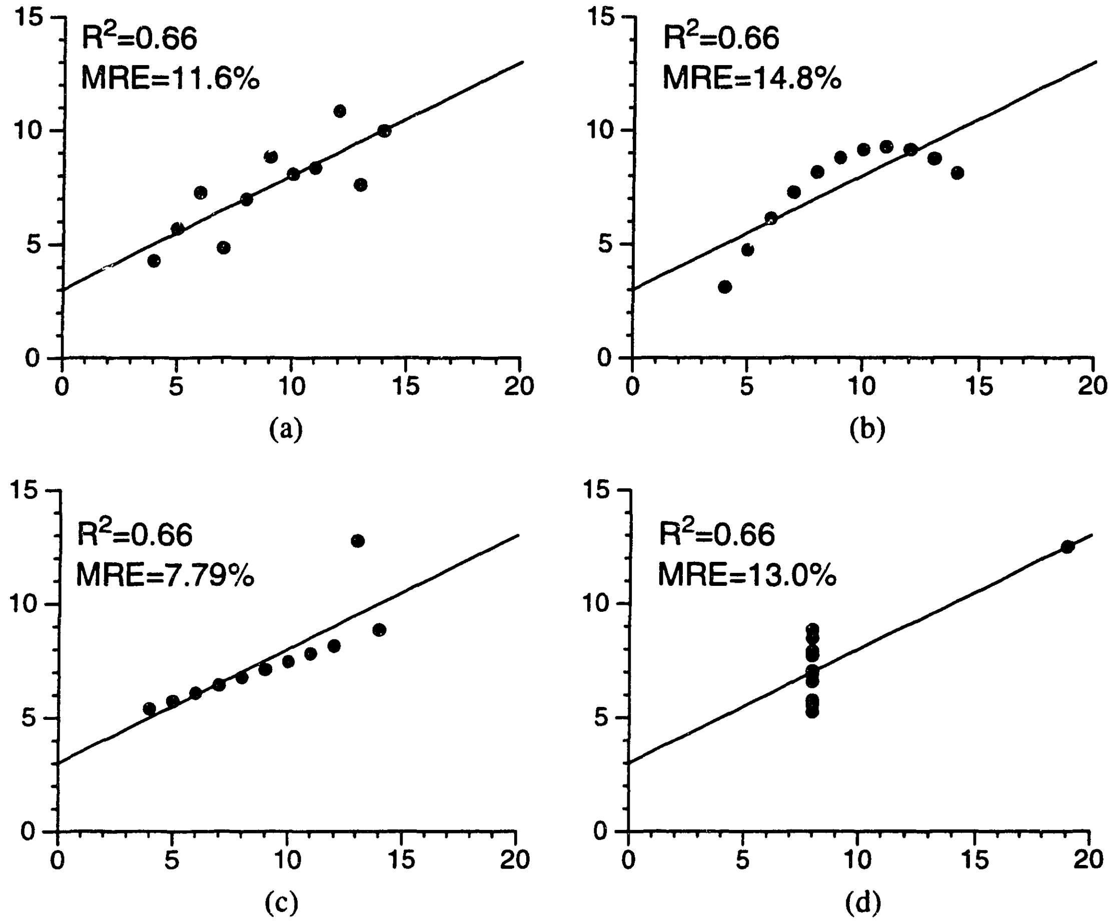


Figure 2-3: Techniques for Model Evaluation

All four data sets lead to identical models and R^2 values, even though the quality of the model varies. The MRE, mean relative error, (defined in Section 2.2.4) quantifies the adequacy of the model with more success. These data sets first appeared in Anscombe [Ans73] and also appear in Hogg and Ledolter [HL92]. We extended Anscombe's plots with the MRE information.

The absolute-value operation ensures that an error of 10% evaluates to $r_i = 1.1$ regardless of whether the prediction is over or under by 10%. The addition of one converts the error value into a multiplier, similar to multiplying by 1.06 to compute a total that includes 6% sales tax. Given the relative errors for a data set with n samples, the mean relative error is the geometric mean of the r_i:

$$MRE \equiv \sqrt[n]{\prod_{i=1}^{n} r_i} \qquad (10)$$

Note that by convention, the geometric mean is used for ratios rather than the arithmetic mean. Figure 2-3 reveals that the MRE gives a more useful evaluation of model quality than the coefficient of determination, but still not as good as the human eye. Determining a specific threshold for a "good" model depends on the situation; the next chapter discusses thresholds for computer-systems models.

2.2.5 Confidence Intervals for Parameters

Given that we can compute the values of the coefficients, we would also like to know how confident we are of each value. Assuming the variance of the error terms is σ^2, one can show that β_1 is a random variable with a normal distribution[1], and that:

$$\text{Variance}(\beta_1) = \frac{\sigma^2}{\sum_i (x_i - \bar{x})^2} \tag{11}$$

In practice, we do not know σ^2 and must estimate it from the data using the following equation:

$$\sigma^2 \approx V \equiv \frac{\sum_i \varepsilon_i^2}{n - m} \tag{12}$$

where n is the number of samples and m is the number of explanatory variables. Given the estimated variance, we can use the t-distribution[2] to compute an α-percent confidence interval for β_1:

$$\beta_1 \pm t(\alpha, n-2) \sqrt{\frac{V}{\sum_i (x_i - \bar{x})^2}} \tag{13}$$

where n is the number of samples and the t function computes the t-distribution given the desired confidence and the number of degrees of freedom.[3]A similar path leads to the confidence interval for β_0:

$$\beta_0 \pm t(\alpha, n-2) \sqrt{V\left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_i (x_i - \bar{x})^2}\right]} \tag{14}$$

1. See Hogg and Ledolter, Section 9.2 [HL92].
2. The t-distribution is very similar to the normal distribution except that it has thicker tails for small data sets. It is useful for relatively small data sets in which the variance and mean are estimated from the data rather than known *a priori*.
3. For n samples and m terms, there are $n - m$ degrees of freedom.

2.2.5: Confidence Intervals for Parameters

These confidence intervals allow us to determine if the $\hat{\beta}$ values are significantly different from zero. For example, if we determined the value of β_1 to be:

$$\beta_1 = 2.5 \pm 4.1 \tag{15}$$

then we would conclude that β_1 is not significantly different from zero, since zero lies within the interval. This provides a tool to determine the relevance of an explanatory variable: a variable is relevant if and only if the confidence interval for its coefficient excludes zero. Once we decide that a variable is irrelevant, we can simply define its coefficient to be zero and recompute the rest of the coefficients under this assumption. This greatly simplifies automated modeling, since it implies that we can have an arbitrary number of irrelevant terms without affecting the quality of the model.

Returning to our fuel-consumption example, we find that the 90% confidence intervals are:

$$\beta_0 = -0.363 \pm 0.38 \qquad \beta_1 = 1.64 \pm 0.13 \tag{16}$$

From these intervals we deduce that β_0 is *not* significantly different from zero, but that β_1 is significant. This leads to the refined model in which the value of β_1 has been recomputed under the assumption that $\beta_0 \equiv 0$:

$$\text{Gallons per 100 Miles} = \beta_1 \cdot \text{Weight} = 1.52 \cdot \text{Weight} \tag{17}$$

Referring back to Figure 2-2 we can confirm visually that forcing the line through the origin affects the quality of the model insignificantly. We can also compare the two models numerically using either R^2 or the MRE:

Model	$\boldsymbol{R^2}$	**MRE**
-0.363 + 1.64 Weight	0.954	5.3%
1.52 Weight	0.948	5.8%

As expected, the removal of β_0 had little effect on the quality of the model. In general, removing irrelevant terms degrades the numerical evaluations of the model by a small amount: we accept this reduction for a simpler model that is typically more robust in practice. Finally, note that the MRE seems to be affected by the change more than R^2. This occurs primarily because we minimize the absolute error when we solve for β_1, but the MRE evaluates the model based on relative error. The next section discusses models that minimize the relative error.

2.3 Generalized Regression

This section extends the simple linear-regression model in several ways. First, we allow more than one explanatory variable. Second, there may be nonlinear terms in addition to simple variables, and third, each sample error value, ε_i, may have a different variance, rather than requiring that all residuals come from the same distribution.

2.3.1 Generalized Multiple Linear Regression

The generalized model extends the simple linear-regression model in two key ways:

1. There may be more than one explanatory variable, and
2. The terms are functions of the explanatory variables.

The full definition is:

$$y_i = \beta_0 f_0(\vec{x}_i) + \beta_1 f_1(\vec{x}_i) + \ldots + \beta_m f_m(\vec{x}_i) + \varepsilon_i \tag{18}$$

Although the model is a linear combination of terms or *basis functions*, it is *not* a linear model: the terms may be nonlinear, which greatly extends the usefulness of these models. The first term is normally defined as $f_0(\vec{x}_i) \equiv 1$ so that β_0 always represents a constant offset. Finally, note that the number of basis functions is independent of the number of explanatory variables.

2.3.2 Weighted Samples

As presented, the extended model has the same "best" solution as the simple model: minimize the sum of the squares of the error terms. However, we would also like to extend the definition of the error terms, so that each sample can have a different weight for its error value. In particular, each sample, y_i, has a corresponding weight, w_i. By convention the error term is inversely proportional to the weight: terms with a large w_i have less impact on the summation. This leads to a new minimization problem:

$$\chi^2 \equiv \sum_{i=1}^{n} \left[\frac{y_i - \hat{y}_i}{w_i}\right]^2 \equiv \sum_{i=1}^{n} \left[\frac{y_i - \sum_{k=0}^{m} \beta_k f_k(\vec{x}_i)}{w_i}\right]^2 \tag{19}$$

The best solution is thus the $\vec{\beta}$ that minimizes χ^2; hence, this is called the *chi-square fit*. There are several reasonable ways to set the weights; for example, if $w_i \equiv 1$ for all i, then we have the standard least-squares model.

A more sophisticated use of the weights is to give samples with high variance less weight; the previous linear-regression model required that all error values come from a normal distribution with the same variance. Thus, if we denote the variance of the i^{th} sample as σ_i^2, then we can define $w_i \equiv \sigma_i$. In this case, the weight of each sample error is inversely proportional to its standard deviation: samples with high variance are given less weight. This version is common in astronomy, where variations in weather affect the reliability (and thus variance) of the measurements. Weighting by standard deviation leads to provably better coefficients.

We use the weights in a novel way to improve the quality of models with a wide range of output values. In particular, we can use the weights to build models that minimize the relative error rather than the absolute error. This is critical for computer-systems models for two reasons: first, we expect the measurements to vary widely for many problems, and second, we cannot tolerate large relative errors, since the errors may accumulate through loops or other forms of repeated invocation. To minimize the relative error, we simply define $w_i \equiv y_i$:

$$\chi^2 = \sum_i \left[\frac{y_i - \hat{y}_i}{y_i}\right]^2 \tag{20}$$

Returning to our fuel consumption example, we can now compute the coefficients that minimize the relative error. This leads to an improved version of the refined model (after verifying that β_0 remains insignificant):

$$\text{Gallons per 100 Miles} = \beta_1 \cdot \text{Weight} = 1.50 \cdot \text{Weight} \tag{21}$$

The MRE is reduced from 5.8% to 5.7% for the new model.

As with the simple model, the solution of the chi-square model can be obtained by setting the partial derivatives (with respect to each β_j) equal to zero. This leads, once again, to the *normal equations*. However, solution via the normal equations is not very robust and thus Section 3.6.2 introduces a second technique, called *singular-value decomposition*, that is used by the auto-calibration toolkit.

2.4 Summary

This chapter covered the statistical techniques required for the models developed in the coming chapters. In particular, we introduced the generalized linear-regression model with weighted samples. This model provides several attractive features:

1. It is a linear combination of nonlinear terms, which allows for arbitrarily complicated surfaces that need not be continuous or even defined over the entire input domain.
2. The coefficients can be computed automatically based on a reasonable amount of input data.
3. Terms can be tested for relevance by investigating whether or not the corresponding confidence intervals include zero. The coefficients can then be recomputed after the

irrelevant terms are removed.

4. The resulting model can be evaluated automatically using the mean relative error. This allows most models to be created and accepted without human interaction. Questionable models can be investigated further via residual plots.

Although most of the material in this chapter is common background information, there are a few novel contributions. First, we defined the mean relative error, or MRE, as a better metric for automatic model evaluation, especially in situations like computer-systems models in which the expected range of measurements is high. Second, we showed how to use the chi-square fit to generate models that minimize the relative error rather than the absolute error. For the models of the coming chapters, the relative error is far more important.

3

Auto-Calibration Toolkit

After reading the previous chapter, it becomes obvious that the process of building and verifying a model is rather long and complex. Unfortunately, this complexity impedes the stated goal of providing statistical models as a powerful and convenient tool for systems designers and implementors. The process becomes even more daunting if we plan to use a large number of models, which is another goal of this work.

This chapter presents the design and implementation of a suite of tools, called the *auto-calibration toolkit*, that hide the complexity of statistical modeling and thus facilitate a large number of models. In particular, these tools convert simple model specifications into complete fitted and verified models, all without any intervention from the user. The toolkit produced nearly all of the models in the remaining chapters, which saved countless hours and provided an increased level of confidence in the models.

3.1 Toolkit Overview

The auto-calibration toolkit consists of several programs that together convert simple model specifications into complete calibrated models. Several aspects of the toolkit make it indispensable for model building; in particular, it provides:

1. A simple input format that describes the range of the explanatory variables and the names and values of the basis functions,
2. A code-generation program that automatically produces the code for collecting samples,
3. A model builder that computes the coefficients based on the specification and the samples, and

4. A model verifier that checks the model against an independent set of samples from the same sample space.

The resulting model consists of only the relevant terms, which allows the designer to be conservative in deciding what the basis functions should be. A fine point is that the automatically generated data-collection program collects *two* sets of samples: one for model fitting and one for model verification. The first set is produced according to loops defined by the specification, while the second set uses random samples selected uniformly over the full range of each explanatory variable.

Figure 3-1 shows a block diagram of the toolkit. The rest of this section walks through the block diagram at a high level; the following sections cover the tools and files in detail. The input to the toolkit is a file that contains a sequence of model specifications, with an implicit second input of the choice of platform for the generated models. Section 3.2 covers the model-specification language. Given these inputs, the toolkit produces two output files, which contain C and perl [WS91] procedures that implement each model.[1] The C procedures can be linked in with applications to provide dynamic model-based prediction. The perl procedures are used by the high-level library compiler for its model-based decisions. Section 3.6.4 covers these output files.

The first tool is the profile-code generator. As shown in the block diagram, it takes two inputs: the model specifications and a template file for the target platform. The primary task of this tool is to generate C code that collects samples for each model and then to embed this code into the template to build a complete data-collection program. The generated program is referred to as simply the *profiling code*. The secondary task of this tool is to forward a list of the basis functions to the model generator; this list is called the *model structure* and is described in Section 3.3.2.

The second tool shown in the block diagram executes the profiling code to produce samples for each model on the target platform. This tool is essentially a *make* file [Man:make] that compiles and runs the profiling code. The output of the execution is a file containing the *model samples*; Figure 3-7 shows an example output file.

The third and final tool is the *model generator*, which is actually a set of three programs. The first one collates the model-structure and model-sample files into separate files for each model. The second program computes the coefficients for the collated models, and the third verifies each model against reserved samples. Section 3.6 covers model generation in detail. The model-collation program completes the process by converting the fitted, verified models into the C and perl programs that form the final output of the toolkit.

The following sections cover each of these tools and files in more detail; the sections are relatively independent and require only the general knowledge provided by this overview.

1. Perl is an interpreted language similar to the shell languages, `sh` and `csh`. Perl's close resemblance to C was a major factor in the decision to use it for the high-level library compiler; in particular, the generated model code is nearly identical in the two languages.

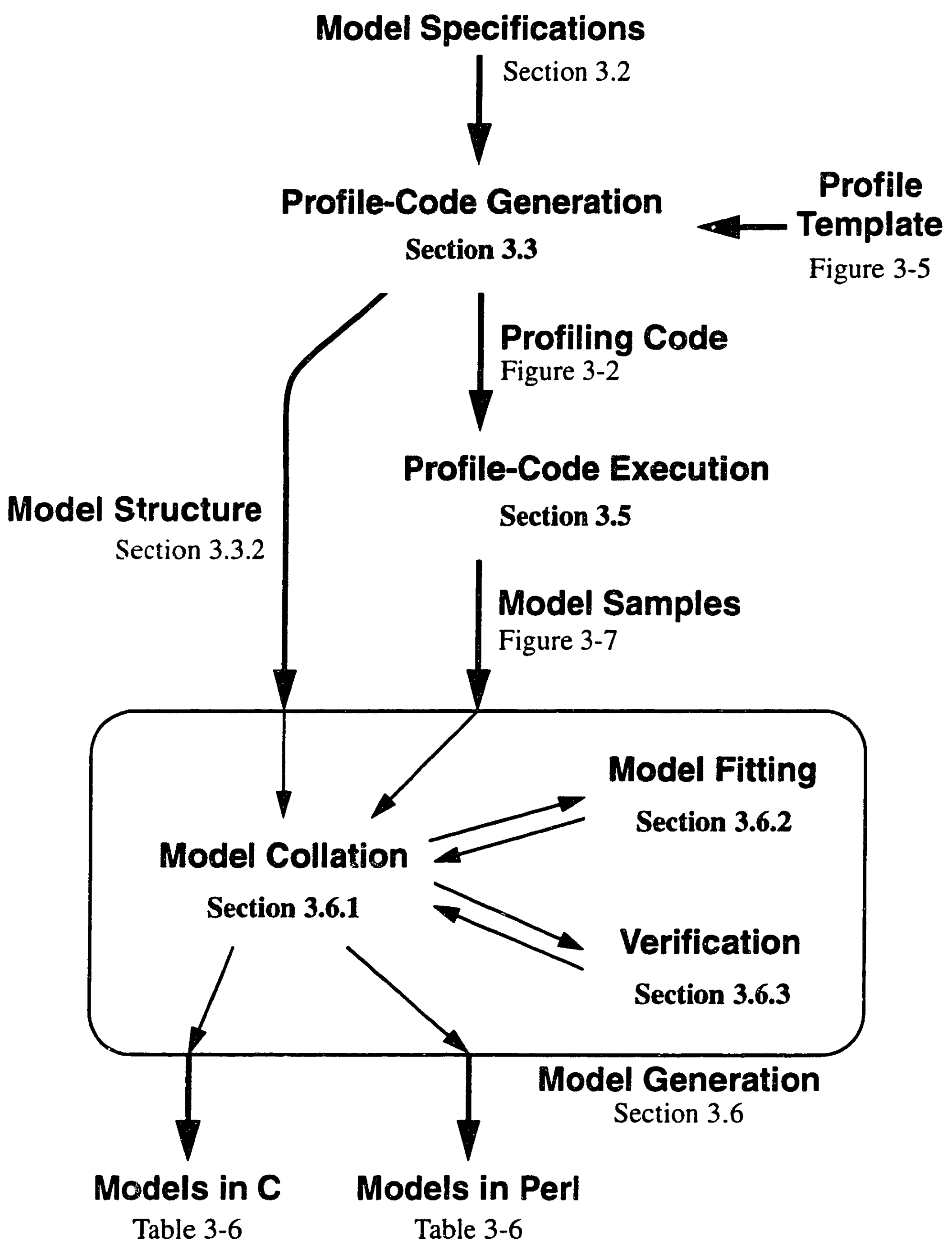


Figure 3-1: Block Diagram of the Auto-Calibration Toolkit

This diagram covers the various steps involved in model generation. The gray boxes correspond to tools (programs), while the arrows correspond to files. The tools and files are covered in sections 3.2 through 3.6.

Component	Description
Task	Procedure to be timed
Loop Nest	Structure for loops surrounding the task
Inputs	The explanatory variables
Terms	Basis functions (excluding explanatory variables)
Setup	Code to execute before the timing
Cleanup	Code to execute after the timing

Table 3-1: Components of a Model Specification

```
Loop according to Loop Nest:
Compute values for Inputs and Terms
Execute Setup code
Start timing
Execute Task
Stop timing
Execute Cleanup code
Output timing and basis-function values to file
end loop
```

Figure 3-2: Pseudo-code for Profiling

3.2 Model Specifications

Before we look at the tools, it is important to understand the model specifications that drive the rest of the system. The goal of the specification language is a simple, unambiguous representation that is both easy for the user and sufficient for the toolkit. The language typically requires about one line per model, with no limit on the number of models per file. Thus, the input to the toolkit is a single file that contains a sequence of model specifications; the output is the corresponding set of generated models, which are calibrated for a particular architecture and evaluated against independent data.

3.2.1 Specification Components

A model specification consists of six components, which are summarized in Table 3-1. These components form the basis of the profiling code and fit together according to the pseudo-code shown in Figure 3-2. The loop nest defines a context in which the code for the task, setup, and cleanup operations executes. In particular, the loop nest defines *iteration variables* that together form the n-dimensional iteration space for the explanatory variables. This context also applies to the code used to compute the values of the explanatory variables and other basis functions. By

Global Identifier	Description
`int *data`	Generic scratch storage
`char *Name`	Name of the current model
`bool Verify`	True when the sample is for verification
`int Select(int a, b)`	Returns a random integer in [a, b]
`void Print(float f)`	Adds a column to the output file
`START`	Macro that starts a timing
`STOP`	Macro that stops a timing
`OUTPUT`	Macro that outputs the current timing
`VCOUNT`	Number of verification samples to take
`int Self`	The id of the current processor
`int PartitionSize`	The total number of processors
`int logP`	The log, base 2, of `PartitionSize`

Table 3-2: Global Identifiers Available to the Profiling Code

convention, explanatory variables are also basis functions, since we can throw out the irrelevant basis functions automatically.

An iteration variable covers an interval of integers that are one-to-one with legal values of the corresponding explanatory variable. For example, an explanatory variable, x, that is limited to powers of two from 2 to 128 is implemented with an iteration variable, i, that covers [1, 7], and a *value function* of:

$$x(i) = 2^i \tag{22}$$

The restriction on iteration variables simplifies the random selection of input values used for collecting validation data, since we can pick any integer in the range. At the same time, value functions decouple the restricted form of iteration variables from the domain structure of the corresponding explanatory variables.

The complete context for the body of the profiling code consists of the set of iteration variables defined by the loop nest, combined with a set of global identifiers that simplify the task, setup, and cleanup operations. Table 3-2 lists the global identifiers.

3.2.2 Input Format

Having covered model specification at a high level, we now take a more detailed look at the input format and the specific syntax and semantics of the model-specification components. Figure 3-3 shows the Backus-Naur form (BNF) of the grammar for the model-specification language. The six components have the following semantics:

<input>	→	<spec>$^+$
<spec>	→	`{{` <loop_nest> `@ Setup @ Task @ Cleanup @` <inputs> `@` <terms> `}}`
<loop_nest>	→	<loop>$^+$,
<loop>	→	`ID Start Stop Update`
<inputs>	|	
<terms>	→	<variable>*,
<variable>	→	`ID : Value`
`Setup`	|	
`Cleanup`	→	*C statement (or list)*
`Task`	→	*C procedure invocation*
`Start`	|	
`Stop`	|	
`Value`	|	
`Update`	→	*Space-free C expression*
`ID`	→	*C identifier*

Figure 3-3: Model-Specification Grammar

This is the BNF for the model-specification grammar. The `typewriter` font denotes terminals, angle brackets denote nonterminals, and the notation "<x>$^+$," denotes a non-empty comma-separated list of <x>.

Loop Nest: The loop nest consists of a sequence of loop definitions, each of which defines one iteration variable. For example, a model with two iteration variables, $i \in [2,6]$ and $j \in [1,100]$, would have the following loop-nest definition:

```
i 2 6 ++, j 1 100 *=10
```

which implies an outer loop in which i covers [2, 3, 4, 5, 6] and an inner loop in which j covers [1, 10, 100]. The fourth field, called the *update* field, is simply a self-referential C operator; for example, the "++" given in the i loop implies that i is incremented each time through the loop. The update function can be more complicated, however, as shown by the "multiply by 10" operator used for the j loop. This decouples the form of the iteration from the size of the domain of the iteration variable, which is one-to-one with the domain of the corresponding explanatory variable. Thus, j is allowed to take any value between 1 and 100, but only three of those values are used for model-building samples. Values other than these three may be selected when we collect data for verification.

Inputs: The inputs for a model are simply the explanatory variables. Each input is a pair that specifies the name of the variable and the C expression used to compute its value from the corresponding iteration variable. An explanatory variable called "length" based on iteration variable i with value function 2^i, has the following definition:

```
length:1<<i
```

Terms: Terms are additional basis functions used by the model. They have the same format as the Inputs and are almost always combinations of explanatory variables. Their value expressions must be functions of the explanatory variables.

Setup: The setup code executes before the measurement to ensure that the system is in the proper state. For example, for tasks that involve multiple processors, it is common to place a barrier in the setup code to ensure that all processors start the timing at the same time (or as close as possible). The setup code can be any legal C statement, which includes sequences of statements and procedure calls.

Task: The *task* for a model specification is the C procedure that encapsulates the action to be modeled. For example, in the clear-memory operation the task is the C procedure call:

```
ClearMemory(data, x)
```

where `data` is the scratch memory area and `x` is a local variable defined as an iteration variable for the number of bytes to clear.

Cleanup: The cleanup code is a sequence of C statements that executes after the measurement to perform any required cleanup and to ensure that the system can continue to the next loop iteration.

These components completely specify the work required to collect samples for each model. The next section examines the automatic conversion of a sequence of model specifications into a complete C program that can be executed on the target platform to generate the model samples required to build and verify each model.

3.3 Profile-Code Generation

Given a file containing model specifications, the generation of the profiling code is relatively straightforward. The tool, called `genprof`, takes two files as inputs, one of model specifications and one containing the *profile template*; the output is the profiling code. The exact command is:

```
genprof <spec-file> <profile-template>
```

The profile template is essentially a header file that defines the macros and global identifiers used by profiling code; Section 3.3.1 discusses the template in more detail.

The basic algorithm for code generation is:

Routine	Description
Append(x, y)	Append list y onto the end of list x
First(x)	Return the first element of list x
Rest(x)	Return all but the first element of list x
Name(*string*)	Return the prefix of *string* up to the first left parenthesis (exclusive)

Table 3-3: Support Routines for the Transformation Language

```
Embed the following into the profile template:
    For each model specification, spec, in the input file {
        Print GenerateModel(spec) to the output file
    }
```

The **GenerateModel**(`spec`) command is the first of a series of transformations that map complete or partial model specifications to strings of C code. Figure 3-4 gives the complete set of transformations.

The transformation language is a simple form of Lisp [Ste90]: the only atoms are strings, so all lists are thus nested lists of strings. The empty list is **nil**, the "+" operator denotes string concatenation, and there is a small set of support routines, shown in Table 3-3. The actual code-generation program is implemented in perl [WS91]; the transformation language presented here should be viewed as a precise specification of the input-output behavior of profile-code generation.

The output of each transformation is a string containing C code that can either be embedded in another transformation or output as part of the profiling code. The best way to understand the transformations is to work through a complete example. Consider the model specification:

```
spec = {{ x 1 10000 <<=1 @ @ ClearMemory(data, x)        (23)
              @ @ length:x @ }}
```

Table 3-4 presents the generated code along with the order in which the lines are generated and the corresponding line number from Figure 3-4. The first step in code generation is the execution of **GenerateModel**(spec), which immediately parses the specification into:

loop_nest = ((`x, 1, 10000, <<=1`))	setup = **nil**
task = "`ClearMemory(data, x)`"	cleanup = **nil**
inputs = ((`length, x`))	terms = **nil**

Second, line 3 (of Figure 3-4) generates header code that places the name of the model into a global variable and indicates that this measurement is for model building rather than verification.

```
GenerateModel(spec) =
        (loop_nest, setup, task, cleanup, inputs, terms) ← spec
        out ← "Name=" + Name(task) + "; Verify=False;\n"
        body ← GenerateBody(setup, task, cleanup, inputs, terms)
        return(out + GenerateLoops(loop_nest, body) +
              GenerateVerify(loop_nest, body))

GenerateLoops(loop_nest, code) =
        if (loop_nest == nil) return(code)
        loop ← First(loop_nest)
        rest ← Rest(loop_nest)
        return(GenerateLoop(loop, GenerateLoops(rest, code)))

GenerateLoop(loop, code) =
        (id, start, stop, update) ← loop
        return("{int "+ id +"; for (" +id +"=" + start +"; " + id +"<="
              + stop + "; " + update + ") {" + code + "}}\n")

GenerateBody(setup, task, cleanup, inputs, terms) =
        out ← ""
        foreach (name, value) in Append(inputs, terms)
              out ← out + "float " + name + "=" + value + ";\n"
        out ← out + setup + ";\nSTART;\n" + task +
              ";\nSTOP;\n" + cleanup + ";\nOUTPUT;\n"
        foreach (name, value) in Append(inputs, terms)
              out ← out + "Print(" + value + ");\n"
        return(out)

GenerateVerify(loop_nest, body) =
        out ← "Verify=True; for (v=1; v<=VCOUNT; v++){\n"
        foreach (id, start, stop, update) in loop_nest
              out ← out + "int " + id +
                    "= Select(" + start + ", " + stop + ");\n"
        return(out + body + "}\n\n")
```

Figure 3-4: Transformations for Profile-Code Generation

Model Specification
`spec = {{ x 1 10000 <<=1 @ @ ClearMemory(data, x) @ @ length:x @ }}`

Generated Code	Step	Line
`Name="ClearMemory"; Verify=False;`	1	3
`{int x; for(x=1; x<=10000; x<<=1){`	10	16
`    float length=x;`	2	22
`    ;`	3	23
`    START;`	4	23
`    ClearMemory(data, x);`	5	23
`    STOP;`	6	24
`    ;`	7	24
`    OUTPUT;`	8	24
`    Print(x);`	9	26
`}}`	11	17
`Verify=True; for (v=1; v<=VCOUNT; v++){`	12	30
`    int x = Select(1, 10000);`	13	33
`    float length=x;`	2	22
`    ;`	3	23
`    START;`	4	23
`    ClearMemory(data, x);`	5	23
`    STOP;`	6	24
`    ;`	7	24
`    OUTPUT;`	8	24
`    Print(x);`	9	26
`}`	14	34

Table 3-4: Map of the Generated Profiling Code

The bottom table shows the output for **GenerateModel**(spec) for the input given in equation (23), which is repeated in the top table. The **Step** column indicates the order in which the lines are generated, while the **Line** column gives the line number from Figure 3-4 responsible for the output. Note that steps 2-9 appear twice because they form the output of **GenerateBody** and are duplicated for the verification code.

Third, we generate the body using **GenerateBody**, which is embedded first into the loop nest, via **GenerateLoops**, and then into the verification code, via **GenerateVerify**.

GenerateBody generates code that defines each of the explanatory variables and terms using the value functions; lines 21-22 emit these definitions. Lines 23-24 then generate the setup code, start-timing macro, task code, stop-timing macro, cleanup code, and finally the macro to output the measurement. In the final step, lines 25-26 generate code that outputs the values of each of the inputs and terms. The work of **GenerateBody** appears in Table 3-4 as steps 2 through 9, which appear twice because the body is duplicated for the verification code.

Given the code for the body, **GenerateLoops** wraps the code for the loop nest around it. This is done in line 12 by recursively descending through the loop nest and then adding the code for each loop as we ascend back up the call chain. This ensures that the loops are generated inside-out: for example, the outermost loop is wrapped around code that includes the inner loops as well as the body. For our example, there is only one loop and the generated loop code appears as steps 10 and 11 in Table 3-4.

The verification code, which is produced by **GenerateVerify**, also consists of a loop wrapped around the body. The first line of the verification code, which corresponds to step 12, sets the `Verify` variable to indicate that the upcoming samples are for verification, and enters a loop to produce `VCOUNT` samples. Lines 31-33 then generate code to pick a random value for each of the explanatory variables; this appears as step 13 in Table 3-4. Finally, line 34 generates a second copy of the body and closes the loop. Thus, the verification code takes `VCOUNT` samples with random (but legal) values for each of the explanatory variables. The value of `VCOUNT` is set by the code supplied by the profile template, which is discussed in the next section.

3.3.1 Profile Template

The profile template provides the context for the profiling code; it defines the global variables, manages the output file, and defines the timing macros. Figure 3-5 presents the profile template for the CM-5 and PROTEUS. First, it defines the global variables `Name`, `data`, and `Verify`. The global variables `PartitionSize`, `Self`, and `logP` are part of Strata and are defined in the included file `strata.h`. Chapter 6 describes the Strata run-time system. The `VCOUNT` macro determines the number of verification samples taken per model; it is currently set to 20.

More interesting are the macro definitions for timing a task. Strata provides a cycle counter that can be used for accurate fine-grain timing. The macro definitions use the cycle-counter routines to perform the timing:

`START`	`start_time = CycleCount()`
`STOP`	`elapsed = ElapsedCycles(start_time)`

The first one copies the value of the cycle counter into `start_time`, which is a local variable defined by the template. The second macro uses Strata's `ElapsedCycles` to compute the difference between the current time and the starting time. The measurement is stored in local variable `elapsed`.

```
#include <stdio.h>
#include <math.h>
#include <strata.h>
#include <sys/file.h>

static char *Name=""; static bool Verify=False;
static int data[10000], data2[10000];
extern int Select(int start, int stop);

#define START start_time = CycleCount()
#define STOP elapsed = ElapsedCycles(start_time)
#define OUTPUT fprintf(out, "\n%s%s %g ", \
    Verify ? "@" : "", Name, CyclesToSeconds(elapsed))
#define Print(x) fprintf(out, " %g", (float)(x))
#define VCOUNT 20

GLOBAL int main(int argc, char *argv[])
{
    int x, y; double g, h; FILE *out;
    unsigned start_time, elapsed;

    StrataInit();
    if (RIGHTMOST) { /* open output file */
        out = fopen("profile.data", "w");
        if (out == NULL)
         StrataFail("Unable to open file profile.data\n");
    }

    << Generated model code goes here >>

    if (RIGHTMOST) fclose(out);
    StrataExit(0);
    return(0);
}
```

Figure 3-5: Profile Template for CM-5 and PROTEUS

This is the profile template used for both the CM-5 and the PROTEUS versions of the profiling code. The generated code for data collection is embedded into this template at the indicated point. The template handles the allocation and definition of the global variables and file management.

The output macros are similar but trickier:

`OUTPUT`	`fprintf(out, "\n%s%s %g ", Verify ? "@": "", Name, CyclesToSeconds(elapsed))`
`Print(x)`	`fprintf(out, " %g", (float)(x))`

The `fprintf` routine outputs values to the profiling data file, which is referenced by `out` in the current context. First, the model name is output, except that it is affected by the `Verify` boolean. In particular, for verification samples only, the model name is prefixed with an "`@`" character. This allows later tools to differentiate model-building samples from verification samples. After the model name comes the timing value, which is converted from cycles to seconds. The `Print(x)` macro outputs a floating-point value preceded by a space; the generated code calls it once for each basis function (input or term).

The generated code of Table 3-4 combined with the profile template shown in Figure 3-5 constitutes the entire profiling code for the `ClearMemory` specification of equation (23). This code is then compiled and run on the target platform to produce the model-fitting and verification data for this model.

3.3.2 Model Structure

In addition to the profiling code, the code-generation tool also produces a list of terms, called the *model structure*, that represents the names of the models and their basis functions. These terms are one-to-one with the columns produced by the profiling code. For example, the model structure for the `ClearMemory` model has two terms `ClearMemory` and `length`, which correspond to the columns for the measurement and the single basis function. Section 3.6.1 and Figure 3-7 discuss the relationship between the profiling data and the model structure in more detail.

3.3.3 Summary

This section covered the transformation of simple model specifications into complete C code for data collection that generates samples for both model building and model verification. The next section covers a variation of the generated code that works in the presence of timing errors, which in turn isolates the rest of the toolkit from issues of data reliability.

3.4 Dealing with Measurement Errors

As described so far, the profiling code assumes that all measurements are valid, which is true for PROTEUS but not for real machines. In particular, the CM-5 suffers from a variety of effects that can ruin samples, so the profiling code must be robust to measurement errors. By dealing with errors during measurement, we isolate the rest of the toolkit from the difficult problems associated with unreliable data. This section examines the sources of timing errors on the CM-5, although it is reasonable to expect these kinds of errors to appear in other multiprocessors as well.

The first class of errors follows from our use of the cycle counter as a fine-grain timer: the cycle counter times all events that occur in the timing interval, which includes timer interrupts and process switches due to time slicing. The second class of errors involves systematic inflation of measured times due to anomalies in the CM-5 operating system. Both classes appear in a CM-5 network study coauthored with Bradley Kuszmaul [BK94].

3.4.1 Incorrect Timings: Handling Outliers

We use the cycle counter because it is extremely accurate: using in-line procedures the overhead can be subtracted out to yield timings that are accurate to the cycle. However, the cycle counter counts everything including interrupts and other processes. For example, if our process is time sliced during a timing, then we count *all* of the cycles that elapse until we are switched back in and the timing completes. Fortunately, the probability of getting switched out during a one-millisecond event is only about 1 in 100, since the time-slice interval is one-tenth of a second. A more common problem is the timer interrupts, which occur every sixtieth of a second and last about 30 microseconds.

These two effects imply several things about the expected errors. First, the measurement is usually correct and very accurate. Second, some fraction of the time the measurement will be over by the time for a null timer interrupt, which is about 30 microseconds. Third, some smaller fraction of the time, the measurement will be significantly too large, often by more than a tenth of a second, since a single time slice usually lasts that long. These kinds of errors are referred to as *outliers*. Clearly, including outliers in the computation of model coefficients could lead to poor models.

Figure 3-6 presents a histogram of 10,000 raw timings for the procedure `ClearMemory(data, 16384)`. Of the 10,000 timings, 91.3% are all within 50 cycles of each other; they represent the correct result. About 5.6% of the time, a timer interrupt ups the timing by about 1000 cycles, which is a 6% error. About 0.18% of the time, the measurement includes a time slice, which gives results that are off by a factor of about 13. Finally, there is a small hump to the immediate right of the main group; these represent measurements that were not interrupted, but were inflated by cache misses due to a recent context switch. Likewise, the tail to the right of the second hump is due to cache misses.

To handle outliers, the profiling codes uses a more sophisticated technique for sample collection. In particular, we use the median of three measurements. Since most measurements are correct, the median is very likely to give us a correct timing. If we assume that an error occurs with probability 10%, then the chance that two measurements are in error is about one in one hundred. In fact, the chance in practice for short events is essentially zero, because the measurements are not independent. Since the timer interrupts occur at a fixed frequency, it is normally impossible for two of three consecutive (short) measurements to be interrupted.

For long measurements, we expect a certain number of timer interrupts and really only want to avoid the time-slice errors. The median is far preferable to the average of three measurements, since the median only produces an error if two of the three timings include time slices, which is extremely unlikely. The average, on the other hand, gives an error in the presence of only a single time slice.

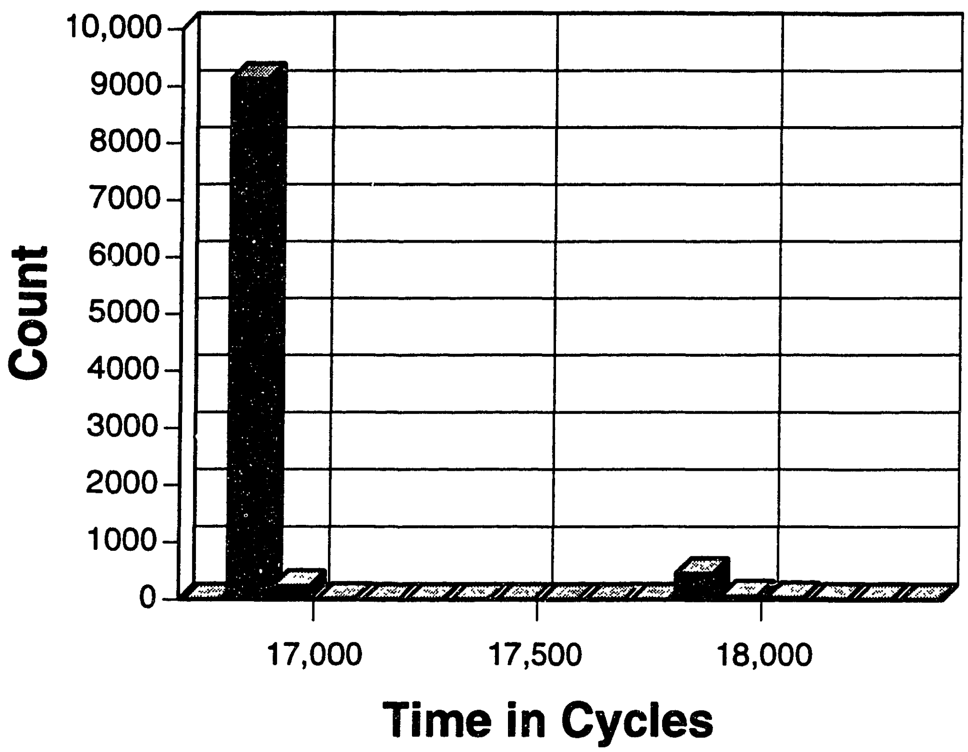


Figure 3-6: Histogram of CM-5 Timings for ClearMemory
Nearly all of the samples are within 50 cycles of each other. However, there is a clump of about 500 around 17,850 cycles and an unshown clump of 18 between 198,000 and 232,000 cycles. (The Scout CM-5 has a clock rate of 33 million cycles per second.)

The median-based sample is implemented by replacing **GenerateBody** with a version that puts the sample code in a loop of three iterations and then outputs the median of the three timings. In fact, when we treat the 10,000 samples used for Figure 3-6 as 3,333 sets of three and take the median of each set, *every single set* produces the correct timing.

3.4.2 Systematic Inflation

For really long timings, those that we *expect* to cover several time slices, the cycle counter is essentially useless. For these models, we use a different template file that uses timers provided by the CM-5 operating system. The alternate macro definitions are:

`START`	`start_time = CurrentCycle64()`
`STOP`	`elapsed = CurrentCycle64()-start_time`

These routines actually come from Strata, which uses the operating system's timers to build a 64-bit cycle counter that only counts user cycles (unlike the hardware cycle counter used by the previous macros).[1] Although this timer is robust to interrupts, it is too heavyweight for short timings: the access overhead is 20-30 microseconds. Thus, the 64-bit counter is appropriate only for timings that are significantly longer than this, so that the overhead is a very small percentage of the measured time.

Although the 64-bit counter solves the time-slice problem, it also introduces a new form of error: systematic inflation of timings that depends on the degree of network congestion. This fact surprised even the authors of the CM-5 operating system and remains unexplained. However, the data is quite clear.

To demonstrate the time-dilation of the operating-system timers, we ran the following experiment. We timed a floating-point loop with the network empty, filled up the network, and timed the same loop with the network full. The only difference between the "empty" time and the "full" time is the presence of undelivered messages sitting in the network. The floating-point code is identical, no messages are sent or received during the timing, there are no loads or stores, and the code is a tight loop to minimize cache effects. The following table summarizes the results for 18 samples taken across a wide range of overall system loads:

Network Status	Average	95% Confidence Interval
Empty	4.56 seconds	±0.0020
Full	5.52 seconds	±0.24

Table 3-5: The Effect of Network Congestion on CM-5 Timings

Not only does filling up the network increase the measured time to execute the floating-point loop by an average of 21%, but it substantially increases the variation in measured time as well, as shown by the wider 95% confidence interval.

This implies that timings that occur while the network is full are inflated an average of 21%. The dilation is load dependent, but we were unable to get a reliable correlation between the inflation and the average system load. Fortunately, the timings appear to be consistent given a particular mix of CM-5 jobs, and the inflation appears to change very slowly with time. Note that algorithms that keep the network full actually achieve lower performance than algorithms that keep the network empty. Thus, the profiling code does not adjust timings for inflation: in fact, we want the models to capture this effect since it is a factor in deciding which algorithms to use.

1. The alternate template defines `start_time` and `elapsed` as 64-bit unsigned integers compared with their previous definition as 32-bit unsigned integers.

We believe that the dilation is caused by context switching. At each time slice, the operating system empties the network, using the "all-fall-down" mechanism[1], so that messages in the network that belong to one process do not affect the next process. When the process is switched back in, its messages are reinjected before the process continues. The cost of context switching appears to depend on the number of packets in the network, and some or all of this cost is charged to the user and thus affects the timings.[2]

3.4.3 Summary

This section looked at two classes of measurement errors on the CM-5. The first source occurs when we use the hardware cycle counter for fine-grain timing. Each measurement has a small probability of being invalidated by an interrupt. The toolkit handles these errors by using the median of three samples to produce one robust sample.

The second class of errors occurs when we use timers provided by the operating system. Although robust to interrupts, these timers are too expensive to use for short timings and suffer from timing inflation that depends on the degree of network congestion. However, since this seems to be a real effect rather than a transient or periodic error, the profiling code includes the inflation in the reported sample.

Finally, it is important to realize that by handling errors at the time of data collection, we isolate the rest of the toolkit from these issues. In particular, the model generator can assume that all of the samples are valid, which greatly simplifies the process and helps to ensure that we get accurate and reliable models.

3.5 Profile-Code Execution

The previous three sections discuss the conversion of model specifications into profiling code that produces reliable samples. Given this code, the next step is to execute the code on the target platform to produce samples for each of the models. In the current design, the profiling code is completely portable, which means that we can generate several data files from the same profiling code, one for each of the target platforms. As an example of an output file, Figure 3-7 gives a fragment from the output for the `CombineVector` model, which is part of the Strata run-time system.[3]

Compiling and executing the profiling code is obviously platform dependent, but is mostly portable. Currently, the entire toolkit is controlled by a single make file [Man:make] that uses gcc

1. The "all-fall-down" mode is a network state that occurs during process switching. In this mode, messages "fall" down the fat tree to the nearest leaf, rather than continuing to their destination. This ensures relatively even distribution of messages among the leaves, which allows the messages to be stored at the leaves until the process resumes.
2. The use of the CM-5's dedicated mode does not eliminate the dilation, although it does provide a more stable measurement environment.
3. The `CombineVector` operation takes a vector from each processor and performs a scan or reduction operation on each element.

```
CombineVector .00005784 1 4 4
CombineVector .00006927 2 4 8
CombineVector .00008853 4 4 16
CombineVector .00013707 8 4 32
CombineVector .00023070 16 4 64
CombineVector .00042423 32 4 128
CombineVector .00084327 64 4 256
CombineVector .00167703 128 4 512
CombineVector .00335283 256 4 1024
CombineVector .00669810 512 4 2048
CombineVector .01339074 1024 4 4096
CombineVector .02677662 2048 4 8192
@CombineVector .00061532 431 4 1724
@CombineVector .00010412 71 4 284
@CombineVector .01598114 11252 4 45008
@CombineVector .01332716 9383 4 37532
@CombineVector .01866778 13144 4 52576
```

Figure 3-7: Fragment from a Profiling Output File

This fragment lists samples for the `CombineVector` model, which has the model-structure entry:

```
CombineVector length logP length*logP
```

The columns of the model structure correspond with the last four columns of the output file; the second column of the output file is the timing in seconds. The lines preceded by an "`@`" are samples reserved for verification.

[FSF93] to compile and link the profiling code. The make files differ only slightly for the CM-5 and PROTEUS, since both platforms use the gcc compiler. The difference occurs in the execution of the profiling code.

On the CM-5, executables are controlled via DJM [Man:DJM], which requires a script for each executable.[1] The make file uses DJM to run the profiling code, with the script hidden as an auxiliary file for the CM-5 version of the toolkit. For PROTEUS, the profiling program is a normal executable, so there is no script file and the make file executes the program directly to produce the output file.

1. The Distributed Job Manager, DJM, manages job allocation for CM-5 partitions. It was developed by Alan Klietz of the Minnesota Supercomputer Center and is widely used as an extension of the CM-5 operating system.

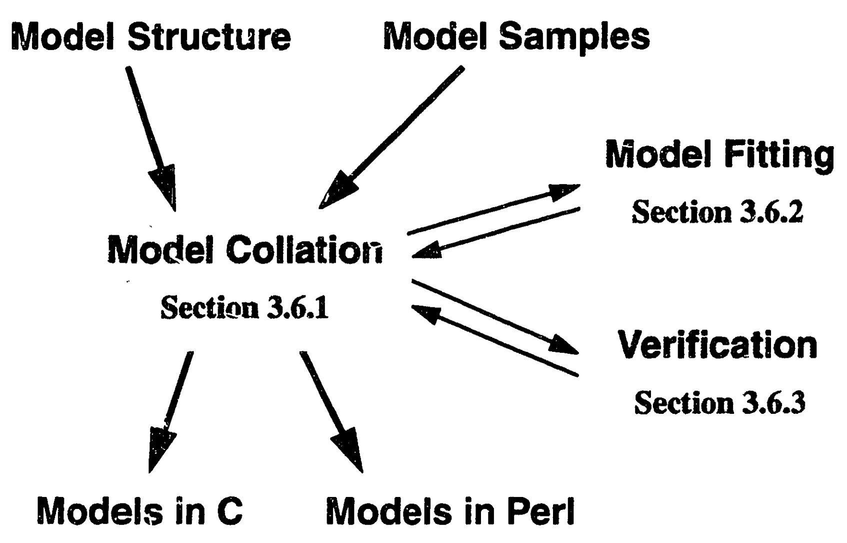


Figure 3-8: Block Diagram of the Model Generator
The model generator consists of three components, repeated here from the complete block diagram of Figure 3-1. Sections 3.6.1 through 3.6.3 cover the three programs, while the output files are discussed in Section 3.6.4.

3.6 Model Generation

The model generator takes the model-structure file and the model-samples files produced by earlier steps and combines them to build and verify each of the models. Figure 3-8 repeats the block diagram of the model generator from Figure 3-1. Given the two input files, each of which cover the same set of models, the model generator produces C and perl code for each model in the set.

The main component of the model generator collates the models into separate files and then invokes subprograms to build and verify models for each of these files. These other two components thus work with files that contain only one model. As each model is built and verified the collation program converts the model structure and computed coefficients into the C and perl models that form that output of the model generator.

3.6.1 Collating Models

The information for each model is split into the two files that contain the same set of models. Each line of the input files is for a specific model that is named by the first field of the line. For example, in Figure 3-7 we see that each of the sample lines for the `CombineVector` model starts with either "`CombineVector`" or "`@CombineVector`". Likewise, the entry in the model-structure file also starts with "`CombineVector`".

To produce the file used for model building, we simply run both files through the Unix utility *grep* [Man:grep] and cull out only the lines that begin with "`CombineVector`".[1] By starting with the model-structure file, we can ensure that the resulting file consists of the model-structure entry followed by the model-building samples. Note that the verification samples are ignored because of the leading "`@`" in the first field.

3.6.2 Model Fitting

The generator repeats this process for every model in the structure file, which produces a sequence of model-building input files. These files are then fed to the model-fitting program, which produces the coefficients for each of the basis functions. The model fitter, called `linreg`, is invoked as:

```
linreg [-r] <model-file>
```

The optional `-r` flag indicates that the coefficients should minimize the relative error rather than the absolute error. The output of linreg contains the coefficients of the basis functions plus the R^2 and MRE values. It also outputs a separate file containing the residuals, which are used to improve the basis functions when the model is inadequate.

Automatic model fitting is an iterative process that repeatedly throws out a single irrelevant basis function until all of the terms are relevant. As discussed in Chapter 2, a term is relevant exactly when its confidence interval excludes zero. The basic algorithm is:

```
Assume all terms relevant
loop {
    Compute coefficients and confidence intervals
    if all terms are relevant then exit loop
    otherwise, remove the least relevant term
}
Output model, $R^2$, and MRE
```

The coefficients are computed by a pair of programs from *Numerical Recipes in C* [PTVF92] that perform χ^2 fitting using a technique called singular-value decomposition, or SVD.[2] The key advantage of SVD over the normal equations presented in Chapter 2 is robustness. The normal equations require that the basis functions be mutually independent, which is a common requirement for linear-algebra problems. If one term is a linear combination of the others, then the normal equations have no solution and any tool that used them would simply give up. The SVD method, however, can detect these situations and adjust. For example, if the same basis function appears twice, then the normal equations will fail, but the SVD solver will give the two terms the same weight. In fact, the total weight of the pair matches the weight you would get if you only had one copy of the term and used either SVD or the normal equations. Thus, SVD is strongly prefer-

1. The exact command is: `grep '^CombineVector'` <structure-file> <sample-file>
2. The two routines are `svdfit` and `svdvar` from Section 15.4, which also covers SVD in more detail.

able simply because it *always* works: there is no set of basis functions that causes unacceptable behavior.

The procedure for computing the coefficients has the following signature:

$$\text{Fit}(\mathbf{F}, \mathbf{Y}, \mathbf{W}, m, n, \vec{R}) \rightarrow (R^2, \vec{\beta}, \vec{C}, \mathbf{E}) \tag{24}$$

where **F** is the matrix of n samples of $m + 1$ basis functions, **Y** contains the n measurements, **W** contains the corresponding weights, and $\vec{R}$ is a vector of $m + 1$ boolean variables that indicate whether or not the corresponding basis functions are relevant:

$$\mathbf{F} \equiv \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & & & & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,m} \end{bmatrix} \qquad \mathbf{Y} \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \qquad \mathbf{W} \equiv \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \qquad \vec{R} \equiv \begin{bmatrix} r_0 & r_1 & \cdots & r_m \end{bmatrix}$$

On output, R^2 is the coefficient of determination, $\vec{\beta}$ contains the $m + 1$ coefficients, $\vec{C}$ contains the variance of the coefficients, and **E** contains the n residuals:

$$\vec{\beta} \equiv \begin{bmatrix} \beta_0 & \beta_1 & \cdots & \beta_m \end{bmatrix} \qquad \vec{C} \equiv \begin{bmatrix} \text{Var}(\beta_0) & \text{Var}(\beta_1) & \cdots & \text{Var}(\beta_m) \end{bmatrix} \qquad \mathbf{E} \equiv \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

The boolean vector, $\vec{R}$, simplifies iteration by allowing us to force the coefficients of irrelevant terms to zero. Alternatively, we could simply remove those columns from **F** and then renumber the elements of $\vec{\beta}$ and $\vec{C}$ on output, but it is much simpler to use $\vec{R}$ to mark the relevant terms as we progress; initially all of the r_i are marked as relevant.

Given Fit we can build models that minimize either absolute or relative error. For absolute error, we simply define $w_i \equiv 1$ for all i. Relative error is a bit more tricky; based on Section 2.3.2, we would use:

$$w_i \equiv y_i \tag{25}$$

However, because the weights can be used to indicate sample variance, larger weights lead to larger confidence intervals. Thus, to maintain the same assumptions about variance as the absolute-error version, we use *normalized* weights:

$$w_i \equiv \frac{y_i}{\bar{y}} \tag{26}$$

Note that the average of the weights is now:

$$\bar{w} \equiv \frac{1}{n}\sum_i w_i \equiv \frac{1}{n}\sum_i \frac{y_i}{\bar{y}} \equiv \frac{1}{\bar{y}}\left(\frac{1}{n}\sum_i y_i\right) \equiv \frac{1}{\bar{y}}(\bar{y}) = 1 \tag{27}$$

which matches the average for the absolute-error case.

From $\vec{C}$ we can compute the half-width of the 95% confidence interval for each of the coefficients. The half-width, h_i, for β_i follows from equation (13) on page 32:

$$h_i = t(95\%, n-r)\sqrt{V \cdot C_i} \tag{28}$$

where t computes the t-distribution given the confidence level and degrees of freedom, r is the *proposed* number of relevant basis functions (initially $r = m + 1$), and V is the estimated variance as defined by equation (12) on page 32. Note that the number of degrees of freedom is $n - r$ instead of $n - m$ since we ignore the columns of **F** marked as irrelevant.

A term is relevant exactly when the confidence interval excludes zero, that is:

$$\beta_i > h_i \quad \text{or} \quad \beta_i < -h_i \tag{29}$$

which is equivalent to:

$$\left|\frac{\beta_i}{h_i}\right| > 1 \tag{30}$$

This ratio thus provides a measure of the relevance for each coefficient. In particular, we define the least relevant term as the one with the minimum ratio. If the least relevant term has a ratio greater than one, then all of the proposed terms are relevant; otherwise, we mark this term as irrelevant and recompute the rest of the coefficients. Figure 3-9 shows the complete model-fitting algorithm.

Once we find a relevant set (or find that none exists), the program outputs the coefficients, using zero for each of the irrelevant coefficients. It also outputs R^2, the MRE, the half widths, and the residuals; these are helpful in case the model fails the next step, which is model verification.

3.6.3 Model Verification

Given the coefficients, the model generator then evaluates the model against the verification data. The verification data is gathered from the sample file via a second grep command; for example, for `CombineVector` we pull out all the lines that start with "`@CombineVector`".

Since the verification lines contain a complete set of values for the basis functions, we can use the coefficients produced by the model builder to compute a predicted timing. We then compute the relative error from the timing and the prediction according to equation (9). Finally, we can compute the MRE from the relative errors of each of the verification samples. If the MRE is

```
Set all $r_i \leftarrow$ True
Loop {
        $(R^2, \vec{\beta}, \vec{C}, \mathbf{E}) \leftarrow \mathrm{Fit}(\mathbf{F}, \mathbf{Y}, \mathbf{W}, m, n, \vec{R})$
        Compute each $h_i$ from $\vec{\beta}$ and $\vec{C}$
        Find $\beta_i$ with minimum ratio, $\rho_i \equiv \left|\frac{\beta_i}{h_i}\right|$, say $\beta_k$
        If $\rho_k > 1$ then exit loop
        Else, set $r_k \leftarrow$ False
}
Output residuals to auxiliary file
Compute MRE
Output $R^2$, MRE, $\vec{\beta}$, and $\vec{h}$
```

Figure 3-9: Pseudo-Code for Automated Model Fitting

greater than 10%, the generator produces a warning about that particular model. Note that we expect the MRE of the verification samples to be higher than the MRE computed by the model builder, since the coefficients were set so as to minimize the relative error for those samples.[1] The verification samples are by design a better test of the model, and thus the MRE returned by the model generator is the MRE of the verification samples.

3.6.4 Output: Models in C and Perl

The last step of model generation is the production of C and perl versions of the model. The C version supports dynamic evaluation of the models, while the perl version forms part of the compiler for high-level libraries. The output code is a sequence of procedure definitions, one for each model, that take the explanatory variables as arguments and return a floating-point number that represents the predicted execution time. Table 3-6 shows the C and perl code produced for the `ClearMemory` procedure.

For C code, the generator first builds a procedure header that defines all of the explanatory variables. The MRE of the verification samples is included as a comment. The body of the procedure is simply a `return` statement that computes the linear sum of the basis functions, using the coefficients from the model builder and leaving out irrelevant terms (those with a coefficient of zero). For the basis functions that are not explanatory variables, the term's value function is used to compute the value of the basis function from the values of the explanatory variables.

1. If the coefficients minimize absolute error, then there is typically a smaller but still noticeable difference between the MRE's of the two sets of samples.

Model Specification
`spec = {{ x 1 10000 <<=1 @ @ ClearMemory(data, x) @ @ length:x @ }}`

Generated C Code

```
double _Eval_ClearMemory(int length) { /* MRE: 1.002 */
   return 57.59 + 1.423 * length;
}
```

Generated perl Code

```
sub ClearMemory { # MRE: 1.002
   local($length) = @_;
   return 57.59 + 1.423 * $length;
}
```

Table 3-6: Generated Code for Model Evaluation

These tables show the generated code for the `ClearMemory` model. The C version can be used at run time for dynamic model-based decisions; it is prefixed with "`_Eval_`" to avoid a name conflict with the actual `ClearMemory` routine. The perl version uses the "local" convention to name the arguments.

The perl version is quite similar. Procedures in perl always take an array of strings as their sole argument, which is denoted by "`@_`". By convention, arguments are named using the `local` command, which takes a list of variables, allocates local storage for them, and assigns to them the corresponding element of the input array. A leading "`$`" denotes identifiers in perl, but otherwise the argument list is the same as for the C code. Likewise, the body is identical except for the addition of "`$`" to all of the identifiers.

3.6.5 Summary

This section described the conversion of model samples into verified models written in C and perl. The model-collation program controls the process and uses the other two components as subroutines to build and verify each model. The model builder can minimize absolute or relative error and automatically removes irrelevant terms from the model. The final coefficients are then verified against independent data using the model-verification program. Finally, the coefficients and model structure are combined to produce the model-evaluation procedures. The final output consists of two files: one containing the C procedures and one containing the perl procedures.

3.7 Summary and Conclusions

This chapter presented the design and implementation of the auto-calibration toolkit. The toolkit brings statistical modeling into the suite of tools available to a system designer. In particular, the toolkit transforms the long and complex process of generating models into a simple process that automatically converts model specifications into calibrated and tested models for the target architecture. The toolkit has several features that are critical to this process:

- ❑ The model specifications are simple and tend to be quite short, usually about one line per model.
- ❑ The toolkit removes irrelevant basis functions automatically. This is critical because it allows the designer to add any terms that he believes *might* be relevant.
- ❑ When there are missing basis functions, indicated by an inaccurate model, the toolkit reveals which of the terms are relevant and generates the residuals, which can be used to identify the missing basis functions.
- ❑ The code for sample collection is generated and executed automatically, thus eliminating one of the more painful steps of statistical modeling.
- ❑ Sample collection is robust to measurement errors, which eliminates another painful aspect of statistical modeling. In particular, the output of the profiling code is a sequence of reliable samples.
- ❑ The toolkit tests the produced model against independent samples to ensure that the model has good predictive power. During recalibration (for a new architecture or a change in system software) usually all of the models pass verification, which means that the amount of human intervention required for recalibration is essentially zero.
- ❑ Finally, the toolkit produces C and perl versions of the models that can be embedded within a larger system, such as the high-level library compiler discussed in the next chapter.

A few key principles drove the design of the toolkit. First, it should automate as much as possible; for the common case of recalibration, in which we expect the basis functions to be correct, the toolkit automates the entire process. Second, the toolkit should be robust to user error. This principle led to the use of confidence intervals to remove irrelevant terms and also to the use of SVD, which behaves reasonably when the user provides basis functions that are not independent. Third, the produced models should be easy to incorporate into a larger system; the production of C and perl models supports both run-time and compile-time use of the models. The upcoming chapters cover systems that use both forms of the models.

The auto-calibration toolkit is unique among tools for system designers and thus provides several novel contributions:

- ❑ A model-specification language that allows a concise description of a model, but contains sufficient information to automate the rest of the process. The description includes the range of the explanatory variables, the basis functions, and a small amount of code that facilitates sample collection.

- A profile-code generator, which converts model specifications into complete C programs for automated data collection.
- An analysis of the sources of timing errors on the CM-5.
- Techniques for sample collection that handle measurement errors and produce reliable samples, which in turn isolates the rest of the system from issues of data reliability.
- A model-fitting program that can minimize either absolute or relative error, and that can methodically remove irrelevant terms automatically.
- A model-verification program that evaluates the produced model against independent data. This acts as a filter on the models that ensures that all of the automatically generated models provide accurate prediction. Models that fail the test are reported to the user for further refinement, but models that are being recalibrated almost never fail.
- A model-generator that converts models into C and perl procedures that can be embedded within a larger system.

3.8 Related Work

There is very little work on automated modeling. There are many tools that can build models from data files. The most popular is certainly *Excel* [Her86]. Other options include mathematical programming languages and libraries. Of course none of these tools help with collecting the data or with converting the models into code that can be used by other tools.

Mathematica from Wolfram Research [Wol91] is a mathematical programming language that certainly has the power to implement the toolkit. However, the statistical package [WR93] provided by Wolfram Research provides only standard least-squares fitting. There is no support for minimizing the relative error or removing irrelevant terms. The other popular symbolic-math system, *Maple* [GH93], has similar restrictions. Similarly, *Matlab* [GH93], a popular tool for linear algebra, provides enough language support to implement the toolkit, but in practice leaves the vast majority of the work to the user. *Splus* [Spe94] is a Lisp-like statistical language with similar power and limitations; the latent power of *Splus* was one of the motivating influences of this work.

Less powerful but still useful are libraries that implement some form of linear regression. The obvious one is *Numerical Recipes in C* [PTVF92], which is actually used internally by the toolkit. The *C/Math Toolchest* [Ber91] is a math library for personal computers that provides some regression routines. Such libraries have even more limitations than the mathematical programming languages, but are much easier to integrate into a larger system. At best these libraries can form part of a larger automated-modeling program.

Another class of related work is tools that convert specifications into programs. The two most famous of these are *lex* and *yacc* [LMB92]. The former takes a specification of tokens based on regular expressions and produces C code for a lexer, while *yacc* converts a BNF-like grammar into C code for a parser that works with the lexer produced by *lex*. The beauty of both programs, as well as the toolkit, is the huge differences in size and complexity between the input specification and the generated code. Note that the toolkit actually contains *two* program generators, one for the profiling code and one for the models.

4

Automatic Algorithm Selection

This chapter develops the concept of *high-level libraries*, which are modules with a particular interface, multiple implementations of that interface, and an automatic method for selecting the best implementation given the target platform and workload. Figure 4-1 summarizes the primary goal of high-level libraries, which is to combine the encapsulation of multiple implementations with automatic selection of the best implementation. This chapter shows how we can use statistical models that are calibrated to the target platform as the basis for automatically selecting the best algorithm or data layout among the available implementations.

We look at two high-level libraries, one for solving partial differential equations via successive overrelaxation that uses a variety of data layouts, and one for parallel sorting of integers that uses versions of radix sort and a sampling sort. For these applications, we show that this technique correctly picks the best implementation more than 99% of the time. In the few cases it made sub-optimal choices, the performance of the selected implementation was within a few percent of the optimal implementation. At the same time, when correct, the benefit was up to a factor of three. Thus, high-level libraries provide a new level of portability, in which the user receives not only a working version, but one with high performance as well.

4.1 Motivation

Traditionally, libraries achieve portability through the portability of their implementation language, which is usually C or Fortran. Unfortunately, such low-level languages require the developer to encode many platform-specific decisions into the source code. Although these decisions are usually made correctly for the initial platform, they are unlikely to remain optimal as the library moves to new platforms or even to new versions of the compiler or operating system.

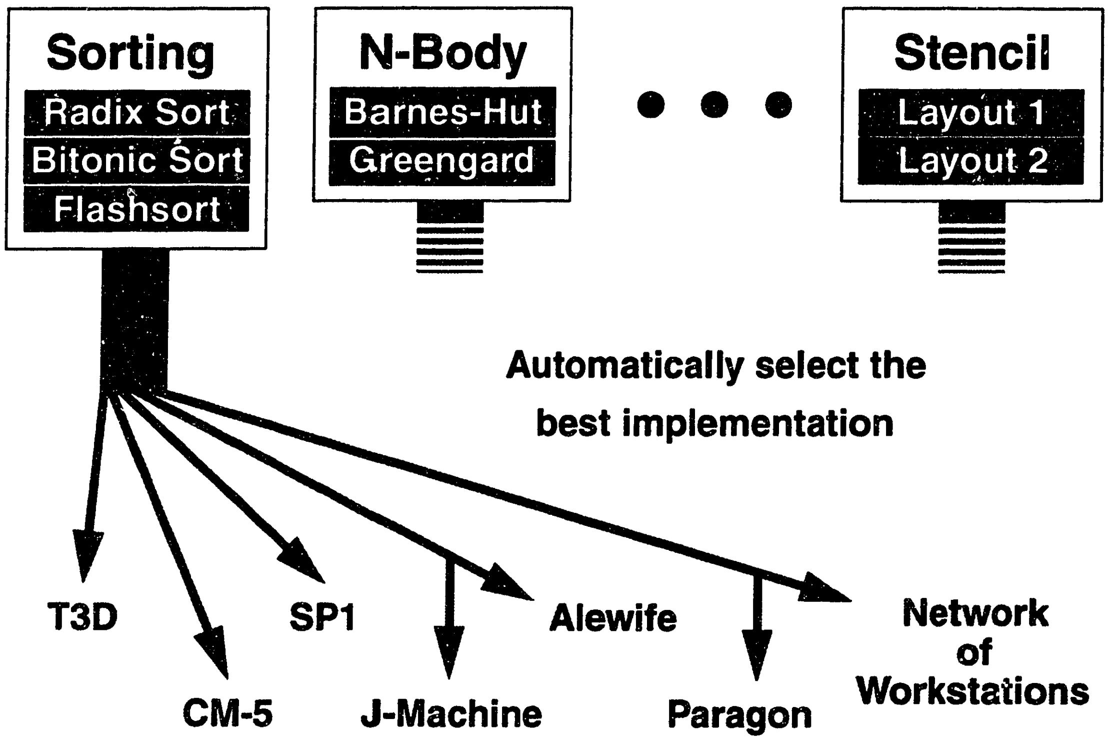


Figure 4-1: The Goal of High-Level Libraries

Conceptually, users have a single view of an application such as sorting or n-body problems. Unfortunately, the wide variety of platforms requires a variety of algorithms to achieve optimal performance. Some applications, such as stencil-based solvers for differential equations, require different data layouts on different platforms for optimal performance. The primary goal of high-level libraries is to select automatically the best algorithm or data layout for the target platform, thus achieving both portability and performance.

This dilemma is particularly painful for supercomputers, because the platforms exhibit a wide variance in the relative costs of computation and communication. This variance greatly reduces the chance that algorithm and layout decisions remain even acceptable as an application moves to a new machine. Figure 4-2 shows the huge variance in costs for common operations on five different architectures. In particular, the cost for barriers can vary by two or three orders of magnitude, the cost for initiating a block transfer by two or three orders, and the per-byte cost by an order of magnitude. At the same time, the CPU performance of the same machines varies far less, with the slowest being the J-Machine [Dal+89], which actually has among the best communication performance.

Furthermore, the demanding nature of supercomputer applications implies that performance losses due to poor choices of algorithm or data layout are not acceptable. In practice, users end up rewriting their applications for each new platform. The assumption in this work is that users rewrite their applications in order to avoid a performance hit that can be an order of magnitude; if

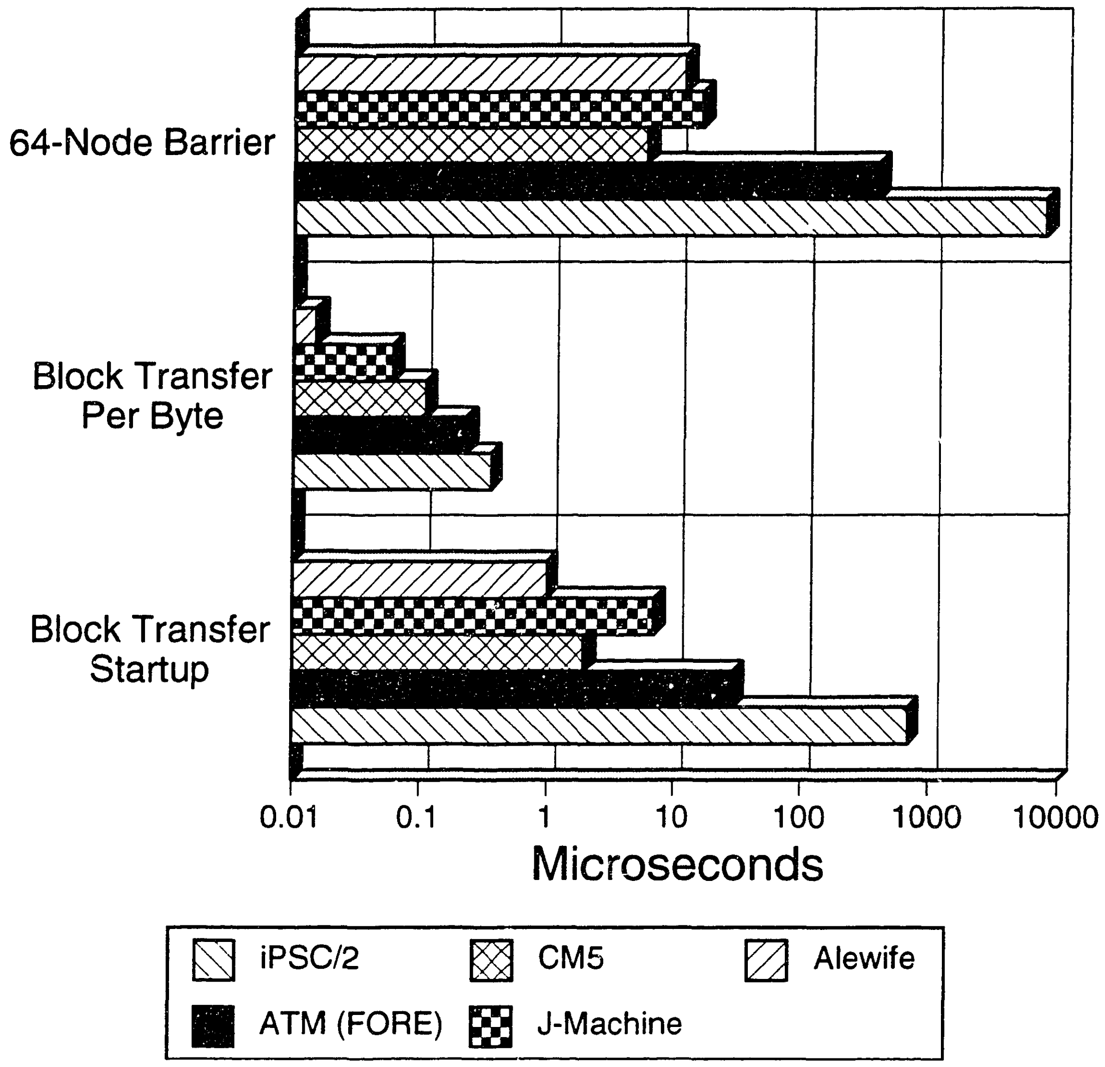


Figure 4-2: The Variety of Costs on Supercomputers

This graph shows the wide variance in the costs of communication. The difference of two to three orders of magnitude in the barrier and block-transfer initiation costs is the fundamental motivation behind high-level libraries. These differences require different algorithms and data layouts on the various machines to achieve optimal performance.

the best implementation of a high-level library brought them close to the best they could do, say within 10-20%, they would be satisfied. By providing several implementations in one library, we greatly increase the chance that at least one of them performs well on each of the user's platforms. In the case where none of the implementations is particularly good, the user still gains: although he must still rewrite the application, the high-level library framework allows the new implementation to be incorporated with the rest, which decreases the need for future rewrites. The framework

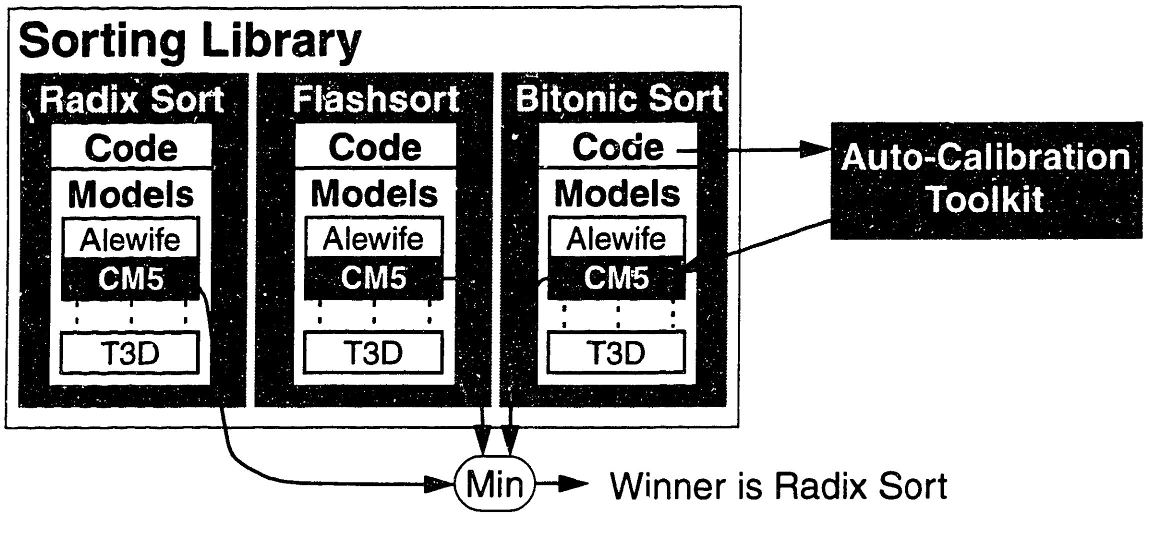


Figure 4-3: Overview of Implementation Selection
The selection of an implementation within a high-level library is based on the models for each implementation, which include one model for each platform. This proliferation of models is not a problem, however, since all of the models are generated automatically by the auto-calibration toolkit. To pick the best implementation for a particular platform, in this case the CM-5, the model predictions for each implementation are compared, which results in the selection of the (predicted) fastest implementation.

also simplifies code reuse over time, over distance, and over institutional boundaries by providing a complete specification and a method for incorporation of new implementations.

4.2 Overview

Figure 4-3 shows the basic process of model-based algorithm selection. In particular, once the calibrated models are available for each platform, the selection is easy: the selector computes a predicted execution time for each implementation and the one with the minimum predicted time is chosen as the winner. The success of this technique clearly depends on the quality of the models; accurate models imply complete success.

Naturally, the models are created automatically by the auto-calibration toolkit, as shown in Figure 4-4. Not surprisingly, a high-level library requires a superset of the inputs required by the auto-calibration toolkit. In addition to the list of terms and the parameter ranges of the independent variables, which are required by the toolkit, the designer must supply the code for the implementations and the interface specification. For the examples in this chapter, the code is in C and the interface specification is thus a C prototype with documentation. Other languages would have their own flavor for these components; for example, languages with better encapsulation mecha-

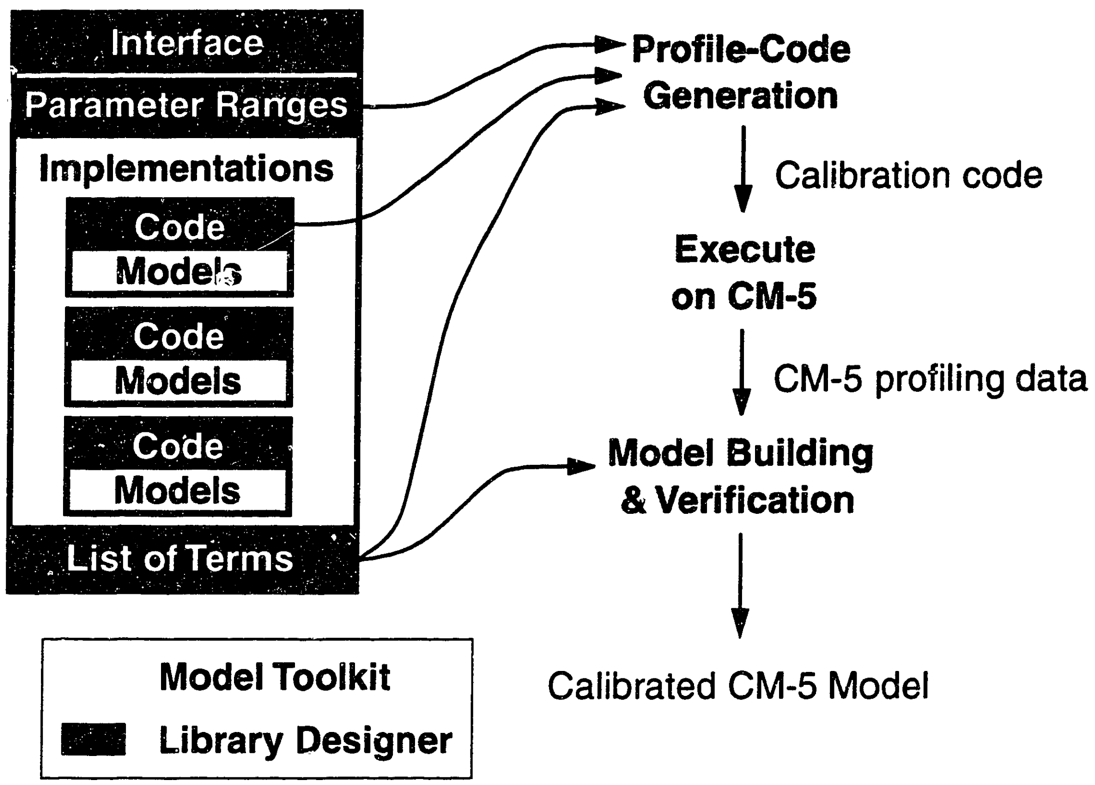


Figure 4-4: Overview of Model Generation

The models used for algorithm selection are automatically generated by the auto-calibration toolkit. This diagram shows the interaction of a high-level library (on the left) with the toolkit (on the right), and highlights which information is provided by the designer and which is generated by the toolkit. As presented in the last chapter, the toolkit handles profile-code generation and execution and the creation of models from profiling data. These models become part of the high-level library.

nisms, such as Modula-3 [Nel91] or Standard ML [MTH90], would use their notion of modules as the basis for both the specification and the implementations.

Although the toolkit produces the models automatically, it is up to the designer to determine when models need to be recalibrated. Obviously, a port to a new platform requires calibration of the models. It is typically desirable to recalibrate the models whenever the environment changes, which includes the compiler, operating system, and the Strata run-time system. The only precondition for recalibration is that the calibration code compiles and executes.

Porting to a new platform thus involves porting Strata and then running the calibration programs for each of the high-level libraries. Porting Strata is not trivial, but it is relatively straightforward and the cost is amortized over all of the libraries, since they only require Strata and C. Occasionally, the list of model terms may have to be extended to capture some unusual behavior

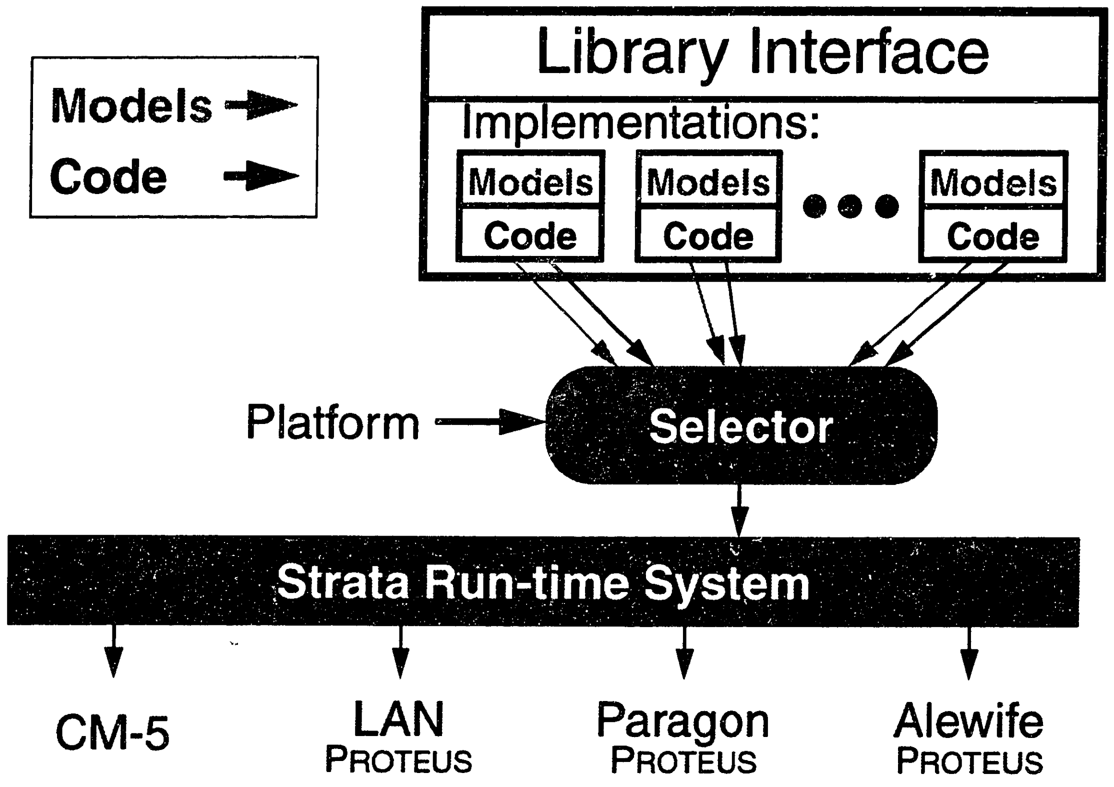


Figure 4-5: Overview of Algorithm Selection

This block diagram shows the components of high-level library selection for the studies presented in this chapter. The selector evaluates the models for each implementation for the target platform, which is one of four architectures. The winning implementation is then compiled for the target along with the Strata run-time system.

of the new platform, but in general the high-level libraries are ready after a single recalibration once Strata becomes available.

4.3 The Platforms

Figure 4-5 presents a block diagram of the components for automatic selection of implementations for this chapter. In particular, there are four target platforms: the CM-5 and simulated versions of MIT's Alewife [Aga+91], Intel's Paragon [Intel91], and a network of workstations based on the FORE ATM switch [FORE92][TLL93]. The three simulated architectures have not been rigorously validated against their actual counterparts, but the Alewife and ATM platforms match their real counterparts on several microbenchmarks. The Paragon version underestimates communication costs, which is due to numbers from Intel that silently assumed kernel-level message passing rather than the user-level communication that would be required in practice. Fortunately, the goal here is not accurate simulation of the platforms, but rather a wide variety of costs that stresses the models' ability to track the architectures.

The Alewife has the fastest communication performance, with the CM-5 a close second. The primary difference is Alewife's support for block transfer via direct memory access (DMA). The Paragon is third with a higher start-up cost for communication, but excellent throughput. The local-area network (LAN) has good throughput, but suffers from very high start-up costs, primarily due to the heavyweight kernel crossing required for each message. The CM-5 alone has hardware support for global synchronization such as barriers. The CPU performance of the four is about the same; by design, the three simulated architectures have the exact same CPU performance. Thus, these four architectures differ in their relative communication and computation costs: the LAN and Paragon favor few large messages, while the Alewife and CM-5 tend to allow many small messages and more communication in general. The models will in fact capture this knowledge precisely without any guidance from the designer, which, of course, is the models' reason for existence.

Having covered the selection process and the platforms, we now turn to particular high-level libraries. First, we examine a library for stencil computations, which are a technique for solving partial differential equations. The key issue for the stencils is the data layout; different layouts require different amounts of communication. The second example is sorting, in which the best algorithm depends on the platform, the size of the problem, and the width of the key.

4.4 Stencil Computations

Stencil computations are a technique for iteratively solving partial differential equations. The basic technique is two-dimensional red-black successive over relaxation [FJL+88][BBC+94]. The word "stencil" refers to the communication pattern: the two-dimensions are discretized into a *virtual grid* and each iteration involves communication with a fixed set of grid neighbors. For example, the basic stencil for successive over relaxation is:

$$x_{i+1} = \omega\left(\frac{1}{4}\text{North} + \frac{1}{4}\text{South} + \frac{1}{4}\text{East} + \frac{1}{4}\text{West}\right) + (1 - \omega)\,x_i \qquad (31)$$

where x_i is the value of the center of the stencil at the i^{th} iteration, the directions refer to grid neighbors, and ω is the extrapolation factor or the degree of overrelaxation. The effect of ω is to speed up convergence by causing the system to overshoot slightly on each iteration. Legal values of ω range from 1 to 2; higher values lead to overshooting by larger amounts. The best ω for our input problems was 1.87.

Conceptually, the stencil is applied to every grid point on every iteration. However, in practice concurrent updates of neighboring grid points can prevent convergence. Thus, the parallel versions use *red-black* ordering, which means that we two-color the grid into red and black regions in which all edges connect one red vertex and one black vertex. For a two-dimensional grid, this is just a red and black checkerboard with the dimensions of the virtual grid. Given the coloring, we break each iteration into two halves: a red half and a black half. In each half, the corresponding nodes are updated by the stencil. This prevents concurrent updates, since red vertices only have black neighbors (and vice versa).

It usually take many iterations for the solver to converge. Convergence is determined by tracking the largest update difference, or *delta*, for each node; if all of the nodes had delta values less than some predetermined convergence threshold, then the system has found a solution.

The virtual grid is partitioned onto the physical processors and then the updates for each half iteration are performed in parallel. Virtual-grid edges that cross processor boundaries require communication, so the choice of data layout greatly affects the performance of the solver.

4.4.1 Data-Layout Options

There are many ways to partition a virtual two-dimensional grid among the physical processors. We would like a partitioning that ensures load balancing and minimizes communication. However, after requiring that these criteria be met, there remain many reasonable choices. In this section, we examine six choices and look at three in detail. Figure 4-6 shows the six choices graphically for a 16-node target.

The simplest layout is to place all of the data on one processor, called the "uniprocessor" layout. This layout has no communication, but only uses one processor; it is primarily useful for small problems that would otherwise be dominated by the communication costs.

The second layout option is the most common one: partition the grid into equal-sized blocks and place one block on each processor. The processors form a square grid; for example, a 64-node machine would use an 8x8 grid. The square grid has the important property that it preserves aspect ratio. For example, an 800x1600 virtual grid leads to blocks that are 100x200 on each node in a 64-node machine. Note that a square virtual grid results in square blocks. The advantage of a square block is that it minimizes the amount of data that must be communicated, since a square has the minimum perimeter for a given rectangular area, and only the elements on the perimeter must be communicated to other processors.

The third option is vertical strips. Each strip covers the full vertical range of the virtual grid, but only covers a small portion horizontally. Thus, a 1600x1000 virtual grid on a 16-node machine leads to 100x1000 blocks on each node. The fourth option, horizontal strips, is similar. The advantage of the strip partitions is that they minimize the *number* of messages, rather than the amount of data. For a two-dimensional grid, the strip versions only need to communicate with two neighbors, while the square-block layout requires most nodes to communicate with four neighbors. For platforms with high start-up costs for communication, we expect the strip versions to perform relatively well, even though they send more data in total than the square version. We examine the vertical-strips version in detail, and discuss the horizontal version at the end of the section.

The fifth option is similar to the square-block layout, but it uses a different aspect ratio. The rectangular-block layout places more processors along one dimension than along the other. In the figure, the ratio is four to one. This layout only wins when the aspect ratio of the virtual grid is skewed. For example, a virtual grid of 1600x200 leads to rectangular blocks of 200x100 and

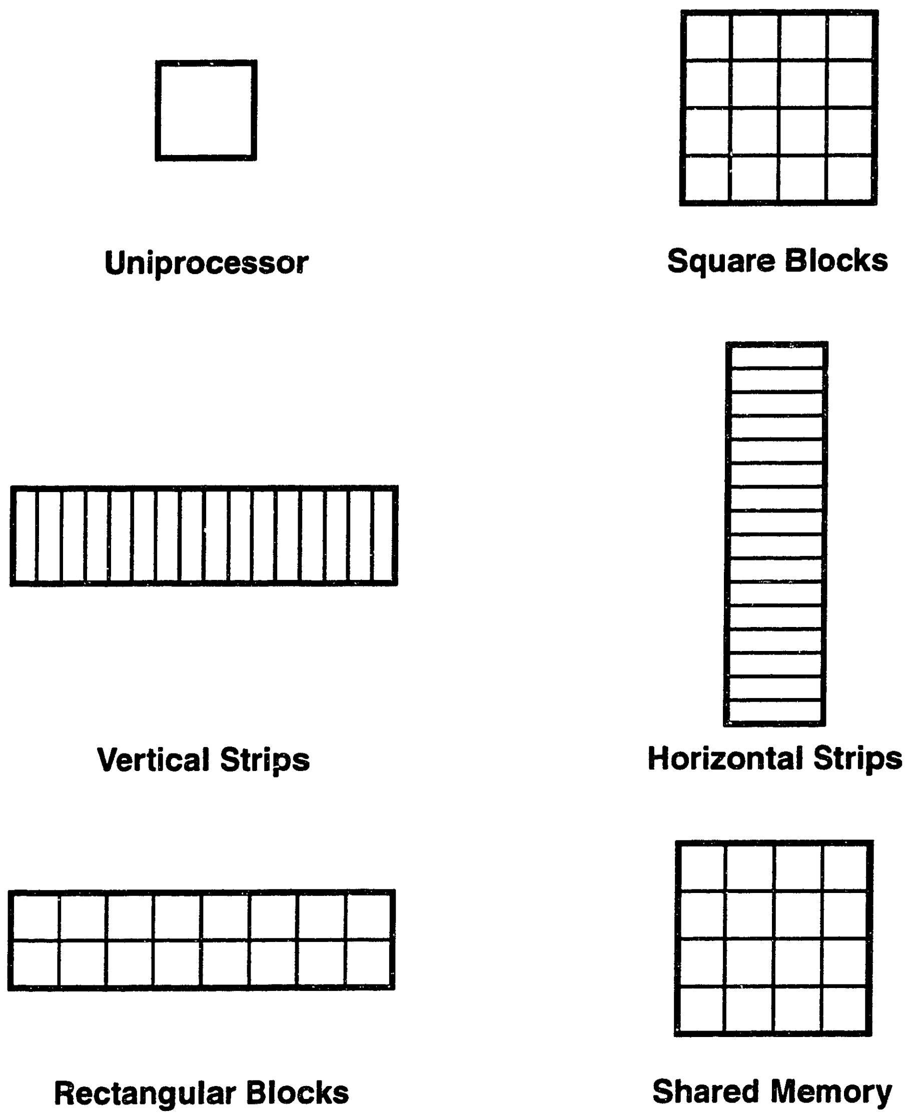


Figure 4-6: Data Layout Options for Stencil Computations
This chapter covers six layout options for stencils. The uniprocessor, square block, and vertical-strip layouts are investigated in detail.

"square" blocks of 400x50. In this case, the aspect ratio of the rectangular blocks compensates for the skew in the virtual grid, which leads to a block size that is closer to a square. Note that there is really a whole family of rectangular partitions, one for each possible aspect ratio. Hence, we will not examine this layout in detail, but we will return to the general issue of aspect ratio at the end of this section.

Finally, the sixth option is to use shared-memory hardware to handle the communication implicitly. The CM-5 does not support shared memory, but the PROTEUS versions can. This is not

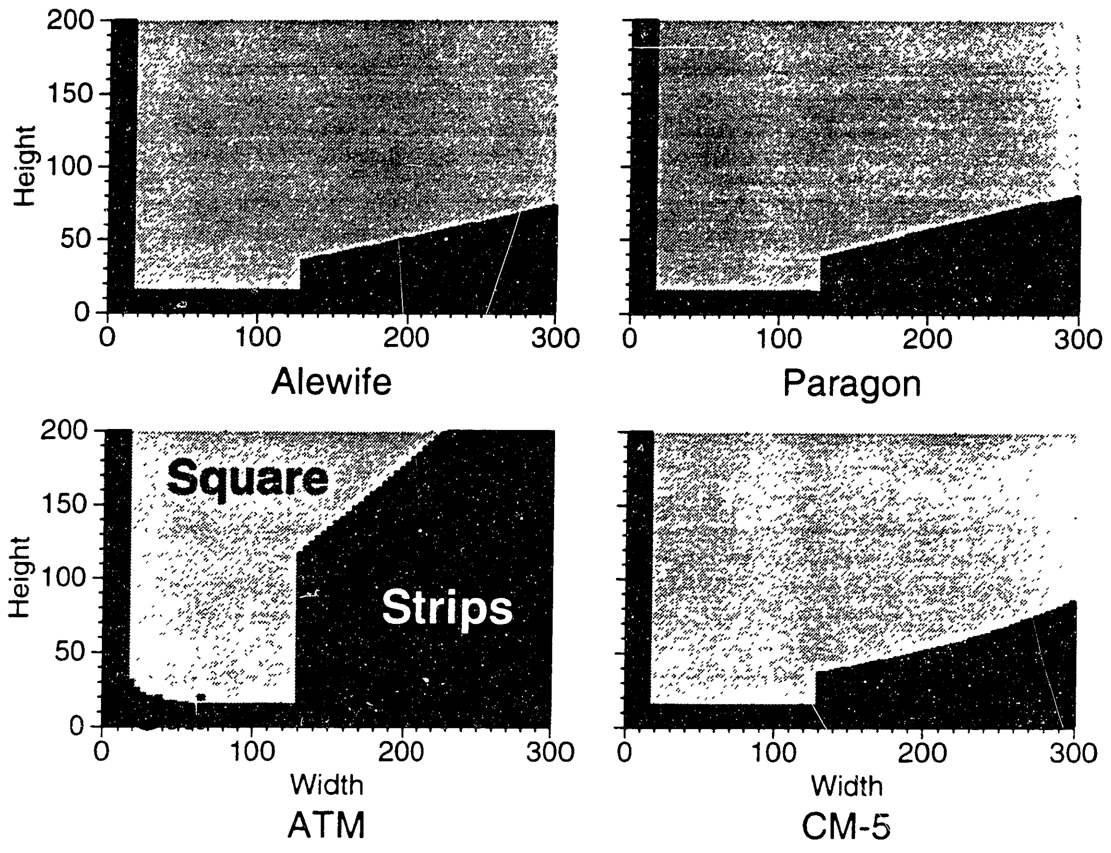


Figure 4-7: Predictions for Stencil Data Layouts

This graph shows the predicted winner for each platform based on the width and height of the virtual grid. As expected, the ATM platform uses the strips version far more due to the high start-up cost for messages on a local-area network. The predictions are quite accurate: the models produce the correct answer more than 99% of the time.

really a layout option, since all of the previous layouts could use shared memory, but it does represent a different implementation and is subject to automatic selection.

4.4.2 Results

Figure 4-7 show the predicted winners for each of the four 64-node platforms for a given width and height of the virtual grid. There are many important facets to this graph. First, the square version seems to be the overall winner, with the vertical strips version coming in second. The strips version wins when the aspect ratio is skewed towards short and wide, which makes sense since each node gets the full vertical range. The strips version wins much more on the ATM network due to the ATM's high start-up cost for communication.

The uniprocessor version wins only for either small width or small height. However, nothing discussed so far explains why the uniprocessor version would win for very tall but thin grids, since such grids encompass a substantial amount of work. This behavior reflects an important aspect of automatic selection, which is that not all of the implementations need to cover the entire input range. In this case, the square version requires a width and height of at least eight, since otherwise some nodes would have zero points. The square implementation could be made to handle this boundary case, but only at a significant increase in the complexity of the code. Instead, the square version simply prohibits part of the input range. The models support this by returning infinity for unsupported inputs. The strips version has a similar restriction: it requires a minimum width of 128 but has no restrictions on the height; the width restriction is clearly visible in the figure.

The models thus provide an elegant way to implement only part of the input space without limiting the end user: as long as each input point is covered by at least one implementation, the selector will always pick a valid implementation. In this case, the uniprocessor version covers the entire space and thus acts as a fall back for inputs that are avoided by the parallel implementations. It is hard to overstate the practical importance of this combination: all of the implementations become simpler because they can avoid parts of the input space that are painful to implement. The use of automatic selection hides this simplification from the user by never selecting inappropriate implementations.

Perhaps surprisingly, the shared-memory version *never* wins. This occurs because nodes alternate ownership of the shared edges. For example, when a processor updates a boundary it pulls the corresponding data to its local memory. Later, when the neighboring node accesses the boundary, it must first pull the data over to its memory. Thus, the shared-memory version requires round-trip messages for every cache line along the boundary. In contrast, the message-passing versions move the data before it is needed and move the entire boundary at once, thus using only one-way messages and amortizing much of the overhead. Conceivably, a better shared-memory protocol could push the data as it is produced, but it would still not match the efficiency of block transfers.[1]

Of course, the figure only shows the predicted regions of victory. If the models are off, the actual regions could look quite different. Thus the key issue is the accuracy of these regions. Fortunately, the predictions are almost always correct, as shown by Table 4-1. Each of platforms achieved a prediction accuracy of about 99%, with the overall average at 99.42%. The smaller machines tended to do slightly worse, primarily because the implementations tend to be closer in performance.

Surprisingly, the CM-5 had the best models, with a noticeable edge over the three simulated platforms. This is presumably due to systematic inaccuracies in the simulations that are avoided by the real machine, but there is no real evidence for this. A known inaccuracy that could have an

1. The DASH architecture provides some protocol support for pushing data [LLG+92]. I actually tried to convince the Alewife group to include such support several years ago, but the end decision was to leave it out until it was more clearly useful. This was the correct decision given their ability to extend protocols in software. Chaiken's protocol tools could implement a protocol for this kind of producer-consumer sharing [CA94].

Platform	16 Nodes	32 Nodes	64 Nodes	Aggregate
Alewife	99.4%	99.5%	99.51%	99.50%
Paragon	99.3	99.5	99.45	99.39
ATM	98.9	99.2	99.21	99.13
CM-5	99.6	99.8	99.71	99.72
Aggregate	**99.3%**	**99.5%**	**99.52%**	**99.42%**

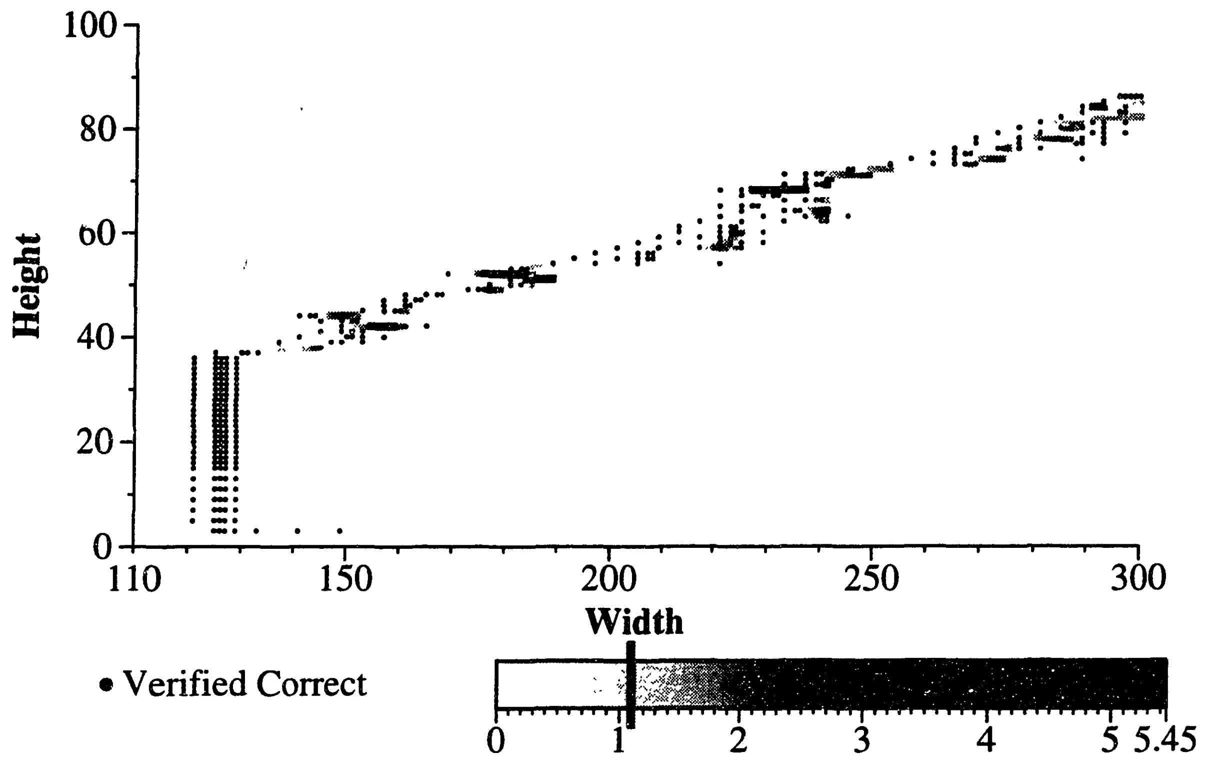
Table 4-1: Prediction Accuracy for the Stencil Library

This table shows the accuracy of the model predictions for a range of platforms and machine sizes; in particular, these numbers reflect the percentage of trials in which the selected implementation was indeed the best. The overall accuracy across all platforms was 99.42%. The 64-node machines have about 3000 data points per platform, compared with only 500 for the smaller machines, and thus provide higher precision.

impact is the artificial delay of interrupts that PROTEUS sometimes uses to increase simulation performance, although there is no evidence that this technique has ever affected the macro behavior of an application.

Each entry in the table represents thousands of executions; the 16- and 32-node numbers represent 500 different virtual grid sizes and the 64-node number represent about 3000 different grid sizes. Each grid size requires four executions, one each layout: square blocks, vertical strips, uniprocessor, and shared memory (for PROTEUS only). Thus, the 64-node Alewife number, for example, represents more than 12,000 simulations, which is why it is accurate to four digits. However, even with this many samples the coverage is spotty: the Alewife prediction graph in Figure 4-7 represents 60,000 grid sizes and thus 240,000 model evaluations. Although 240,000 model evaluations only took minutes, even 12,000 executions is pushing the limits of the available simulation cycles and CM-5 time. To increase the accuracy of the estimates for the 64-node machines, we concentrated most of the selected grid sizes along the boundaries in the prediction graph. This leads to a conservative estimate of prediction accuracy, since we expect most of the errors to occur near the boundaries.

In fact, we can actually plot where the errors occur to see the correlation between the errors and the boundaries of the prediction graph. Figure 4-8 shows the error plot for the CM-5; the other platforms looks nearly the same. The key point is that the errors only occur at the boundaries. If we assume that the models predict execution time to within 2%, then the selector can only make an error at a particular grid size if that size has two implementations whose performance is within 4%. Such close performance is quite rare and really only occurs very close to the intersection of the two model surfaces (which define the boundary).



Error Behavior: CM-5

Figure 4-8: Correlation of Prediction Errors with the Predicted Boundaries
This graph shows where the prediction errors occur in terms of the grid size. The errors form a pattern that exactly tracks the predicted boundary as seen in the CM-5 graph from Figure 4-7. This agrees with intuition since we know the boundaries represent the intersection of the model surfaces for different implementations. For example, if the models are accurate to 2%, then prediction errors can only occur where the surfaces are within 4%, which only occurs near the boundaries. The vertical bar on the legend denotes the mean error when wrong.

The correlation of errors and boundaries has a very important benefit: when the selector is wrong, it picks an implementation whose performance is near optimal. For example, if the actual execution-time surfaces must be within 4% for the selector to make an error, then the performance penalty for that error can be at most 4%. Thus, in the few cases where the selector is wrong, we can take solace in the fact that it will always pick a good implementation. Table 4-2 presents the performance penalties incurred due to incorrect selections. The average penalty when wrong is only 2.12%. If we weigh the penalty by the probability of selecting incorrectly, then we get the expected penalty, which is only 0.0114%. That is, the cost of using model-based selection compared with perfect (oracle) selection is that the library runs 0.01 percent slower on average.

Platform	Mean Penalty When Wrong	Expected Penalty	Worst-Case Penalty
Alewife	2.88%	0.0141%	16.25%
Paragon	3.09	0.0170	13.63
ATM	1.43	0.0113	7.86
CM-5	1.10	0.0032	5.45
Aggregate	**2.12%**	**0.0114%**	**16.25%**

Table 4-2: Performance Penalties for Incorrect Selections

This table reveals the penalty for selection errors for the 64-node machines. The second column gives the average penalty *when the wrong implementation is selected.* If we weight the penalty by its probability from Table 4-1 then we get the expected penalty, shown in the third column. The fourth column gives the worst penalty seen over all of the runs.

Platform	Net Gain
Alewife	8%
Paragon	21%
CM-5	11%
ATM	90%

Table 4-3: Net Gain from Automatic Selection

These numbers reflect the performance gain over always using the square-block layout, which is the best overall. They are rough estimates based on random samples. The ATM platform shows substantial gain because the strips version often beats the square-block layout.

Of course, this number totally neglects the benefit of correct selection! We can not really use the same data to compute the expected gain, since we intentionally concentrated our samples along the boundary. However, we can compute gains from random samples from the full sample space. Unfortunately, it is not clear what the sample space should be, since there is no generic limit to the width and height of the virtual grid. Furthermore, the best strategy in the absence of automatic selection is to pick the best implementation overall, which in this case is the square-block layout. Thus, the measure of gain should be the average difference between the selected implementation and the square-block implementation. Table 4-3 shows the average gains according to this metric for grids up to 5000x5000.

The ATM version shows significantly more gain simply because the square-block layout wins a smaller proportion of the time. These summary numbers reflect 20 samples and should not be given too much weight, since it is not clear that the sample space reflects the interest of a typical end user. Individual samples showed gains ranging from close to zero all the way up to 291% for a skewed grid on the ATM network. Note that these numbers reflect the possibility of incorrect selections, but none of the 80 samples included an incorrect selection (which matches our expectations). Furthermore, we have already established that the impact of incorrect selections is only about 0.01%. To summarize, the stencil library achieves performance gains ranging from 8-90% because of its ability to automatically select among multiple implementations, and it always selects implementations that are optimal or near optimal.

4.4.3 Models

One of the untouched issues is the complexity of the models, which affects the library designer's ability to build accurate models. This section presents the actual models for the 64-node versions of the platforms. Figures 4-9 and 4-10 present the models for the four platforms and the three primary data layouts.

Although none of the models are extremely complicated, there seem to be quite a large number of terms. Part of this complexity results from the fact that the models prohibit parentheses: the terms must be linear combinations. The underlying equation that led to these terms is:

$$\text{time}(\text{width}, \text{height}, \text{iter}) = (a + b \cdot \text{iter})(c + d \cdot \text{width} + e \cdot \text{height} + f \cdot \text{width} \cdot \text{height}) \quad (32)$$

That is, we expect the execution time to have a fixed overhead plus a per-iteration overhead, and that each of these overheads depends on the width and the height, which are represented by the right parenthesized expression. When we flatten this equation into a linear combination, we get:

$$\begin{aligned}\text{time}(\text{width}, \text{height}, \text{iter}) &= ac + ad \cdot \text{width} + ae \cdot \text{height} + af \cdot \text{width} \cdot \text{height} \\ &\quad + bc \cdot \text{iter} + bd \cdot \text{iter} \cdot \text{width} + be \cdot \text{iter} \cdot \text{height} \\ &\quad + bf \cdot \text{iter} \cdot \text{width} \cdot \text{height}\end{aligned} \quad (33)$$

We use this list of terms for the models, except that we give each term an independent coefficient. The use of independent coefficients is required by the toolkit, but also increases the number of degrees of freedom, which increases the power of the model.

The structure of the models follows directly from our high-level understanding of the application. After translating this knowledge into a linear combination of terms, we have at least a good starting point for the list of terms. In this case, the starting list was sufficient and the toolkit returned simple and accurate models.

4.4.4 Conclusions

The performance of the stencil application strongly depends on the layout of the virtual grid onto physical processors. We looked at six implementations and investigated four layouts in detail: square blocks, vertical strips, uniprocessor, and shared-memory with square blocks. The

```
float uni(float width, float height, float iter) { /* MRE:6.839 */
      return (249.7*iter -31.09*iter*width -31.35*iter*height
          +4.1*iter*width*height);
}

float strips(float width, float height, float iter) { /* MRE: 2.341 */
      if (width < 128) { return 1e30; }
      return(6.919 +46.42*height +77.91*iter +12.16*iter*height
          +0.04393*iter*width*height);
}

float square(float width, float height, float iter) { /* MRE: 1.004 */
      if (width < 16 || height < 16) { return 1e30; }
      return(9.04 +6.186*width +5.478*height +123.1*iter
          +2.716*iter*width +1.205*iter*height
          +0.04406*iter*width*height);
}
```

Alewife Models

```
float uni(float width, float height, float iter) { /* MRE: 3.968 */
      return(0.347*width*height +3.47*iter*width*height);
}

float strips(float width, float height, float iter) { /* MRE: 1.672 */
      if (width < 128) { return 1e30; };
      return(0.02343 +0.247*width +1.978*height +0.2343*iter
          +2.47*iter*width +19.78*iter*height
          -0.01034*iter*width*height);
}

float square(float width, float height, float iter) { /* MRE: 1.130 */
      if (width < 16 || height < 16) { return 1e30; };
      return(34.93 +0.3777*width +0.1941*height -0.002268*width*height
          +349.3*iter +3.753*iter*width +1.941*iter*height
          +0.02131*iter*width*height);
}
```

CM-5 Models

Figure 4-9: Stencil Models for Alewife and the CM-5

square-block layout was the best overall, but the uniprocessor and vertical-strip versions also had regions in which they were the best choice.

The shared-memory version never won. It involves round-trip communication rather than one-way communication, and is less efficient for large blocks than using block transfers directly. The code for the shared-memory version was not simpler either, since the allocation and synchronization requirements for good performance made up for the lack of explicit message passing. In fact, the shared-memory code looks very similar to the message passing code.

```
float uni(float width, float height, float iter) { /* MRE: 4.176 */
      return(231.2 * iter -29.45 * iter*width -29.56 * iter*height
          +3.98 * iter*width*height);
}

float strips(float width, float height, float iter) { /* MRE: 1.989 */
      if (width < 128) { return 1e30; }
      return(10.04 +4.867 * width +118.2 * iter -0.4669 * iter*width
          +13.82 * iter*height +0.05118 * iter*width*height);
}

float square(float width, float height, float iter) { /* MRE: 1.403 */
      if (width < 16 || height < 16) { return 1e30; }
      return(17.1 +9.115 * width +5.079 * height +172.6 * iter
          +3.013 * iter*width +0.9967 * iter*height
          +0.04531 * iter*width*height);
}
```

Paragon Models

```
float uni(float width, float height, float iter) { /* MRE: 4.285 */
      return(247.8 * iter -28.96 * iter*width -28.61 * iter*height
          +3.932 * iter*width*height);
}

float strips(float width, float height, float iter) { /* MRE: 1.703 */
      if (width < 128) { return 1e30; }
      return(27.71 +9.097 * width +84.28 * height -0.2605 * width*height
          +282.7 * iter -0.6255 * iter*width +15.86 * iter*height
          +0.06819 * iter*width*height);
}

float square(float width, float height, float iter) { /* MRE: 1.366 */
      if (width < 16 || height < 16) { return 1e30; }
      return(59.25 +31.44 * width +13.96 * height -0.1491 * width*height
          +589.4 * iter +15.08 * iter*width
          +0.0621 * iter*width*height;
}
```

ATM Network Models

Figure 4-10: Stencil Models for Paragon and the ATM Network

The horizontal strips version should work about as well as the vertical strips version, except for blocks that are tall and thin rather than the short and wide blocks preferred by the vertical strips implementation. However, the current implementation of horizontal strips actually performs far worse than the vertical strips version for equivalent problems. This occurs because the array is laid out vertically in both versions. That is, each column consists of contiguous memory, but each row is strided. Thus, the vertical strips version uses a single block transfer to move the entire boundary region, while the horizontal strips version must use a strided block transfer, which is significantly slower. Thus, we concentrated on the vertical strips version for further investigation. Although not trivial, the horizontal version could be rewritten to use a layout with contiguous rows, in which its performance should mirror that of the vertical strips implementation. Note that the models capture the behavior of both strips versions quite well despite these issues.

The sixth version is the rectangular-block layout, which is actually a family of layouts. In general, the right approach is to always use a rectangular-block layout, but to adjust the aspect ratio to the one with the best performance. In fact, the square block, vertical strip, and horizontal strip layouts all share the same code with different aspect ratios. The vertical strip version, for example, is really a 64x1 rectangular block layout; it communicates in only two directions exactly because every node is on the top and bottom boundaries. Thus, these three implementations form a group of alternate aspect ratios, among which the selector chooses the best ratio for the current platform and workload. In general, the aspect ratio should match that of the virtual grid, since that leads to relatively square blocks on each processor. The models capture this information implicitly, along with additional information about the exact value of eliminating entire communication directions as is done by the two strips versions.

The models for all of the implementations were quite accurate, which led to picking the best implementation more than 99% of the time on all platforms. For the few cases in which a suboptimal implementation was selected, the selected implementation was nearly as good as the best choice: only a few percent slower on average. We showed that the system only makes errors near the boundary of the performance surfaces, from which it follows that the penalty for these errors is small.

In contrast, the benefit of picking the right implementation was often more than a factor of two, and averaged 8-90% over a random sample of inputs, with the biggest gains coming on the ATM platform due to its high communication cost.

Finally, an unexpected benefit was the tremendous simplification of the implementations that arose from the ability to ignore painful parts of the input range. By adding preconditions into the models, we ensure that the selector never picks inappropriate models. Thus, an immediate consequence of automatic selection is the ability to combine several simple algorithms that only implement part of the input range.

4.5 Sorting

For the second high-level library, we look at parallel sorting, for which there are many algorithms. The goal here is to select the fastest algorithm for the current platform and workload. We look at two algorithms: radix sort and sample sort. Both implementations are based on a sorting study for Thinking Machines' CM-2 by Guy Blelloch *et al.* [BLM+92].

Radix sort requires keys, such as integers, that can be broken into a sequence of digits, and then sorts the data for each digit starting with the least-significant digit. When each of the digits has been sorted, the entire data set is sorted. Radix sort tends to be simple to implement and corre-

```
for each digit from least to most significant do:
      Count the number of each numeral locally
      Compute the global starting (node, offset) pair
           for each numeral
      Transfer each key to its (node, offset) pair
      Barrier for completion of data reordering
end
```

Figure 4-11: Pseudo-code for Radix Sort

spondingly fast. However, it requires rearranging the data for each digit, which can be very expensive.

The second algorithm, sample sort, uses statistical sampling to guess where the boundaries should be for each node, i.e., which keys should end up on each node. If it guesses correctly, then the data can be moved to the correct node in only one pass; the data is then sorted locally on each node to complete the parallel sort. However, there is a chance that too many keys will be sent to one node, in which case that data must be forwarded to somewhere else temporarily. This leads to an adjustment of the keys, followed by another data movement in which almost none of the data actually has to move. Finally, the data is sorted locally to complete the algorithm. Sample sort is complicated to implement, but is extremely efficient in terms of data movement and thus is an excellent choice for platforms with expensive communication.

The models are again very accurate, correctly predicting the best algorithm more than 99% of the time. The penalty for incorrect selection is less than 1%, because the errors only occur near the crossover points from one algorithm to another. The benefit for correct selection can be more than a factor of ten, but is much smaller for most of the input range: 50% would be typical. Overall the sample sort is the best choice, but radix sort does significantly better for small problems or small key widths, and is also very competitive on the CM-5.

4.5.1 Radix Sort

Radix sort is a very old sorting algorithm that is probably most famous for sorting punch cards by hand. After you accidently drop the cards into a heap on the floor, you sort them by the least-significant digit into ten piles and them stack the piles in order with the zero pile on top. After repeating this for each digit, the cards are sorted.[1] There were also card-sorting machines that automated this algorithm. An overview of radix sort appears in *Introduction to Algorithms* by Cormen, Leiserson, and Rivest [CLR90].

Figure 4-11 gives pseudo-code for parallel radix sort. The main loop covers each of the digits, normally a particular set of contiguous bits from a larger integer key. For example, a 32-bit inte-

1. Although its name is unknown to the young practitioners, radix sort is also the algorithm of choice for sorting baseball cards. I remember being amazed when someone showed me this trick for sorting the cards. (Sorting the cards allows you to figure out which cards you are missing.)

ger can be thought of as 8 4-bit digits, 4 8-bit digits, or even 3 11-bit digits. Increasing the width of a digit reduces the number of loop iterations, but increases the work within each iteration. The next chapter examines how to set the digit size optimally and automatically for the target platform.

The first step for each digit is to count the number of keys with each digit value. For example, for base-four radix sort, we would count the number of keys with each of the four possible digits. Thus, the result for the first step for a base-r radix sort is a set of r counts, one for each numeral.

In the second step, we determine the numeral counts globally by combining all of the local counts. This is done using a vector-scan operation, as provided by Strata, with a vector size of r words.[1] The result of the scan is that each node knows the total number of keys to its left for each numeral. For example, if there 40, 50, and 60 keys with numeral zero on nodes 0, 1, and 2 respectively, then the scan would reveal that there are 150 keys to the left of node 3; we refer to this count as the *offset*. Conceptually, if we had a target buffer for each numeral, then node 3 knows that it can store its numeral-zero keys starting at position 150 and continuing for however many keys it has. This mechanism partitions the target buffer into segments that are exactly the right size for each node.

At the same time, we also compute the total number of each numeral, using a vector reduction operation. At this point, each processor knows both the total count for each numeral and the offset it should use for each numeral. We can combine these two to build our aforementioned buffers out of contiguous memory. For example, assume that we have a large array with elements numbered from one. If there 1000 keys with numeral zero, then we say that the first 1000 elements form the target buffer for numeral zero; numeral one would start at element 1001. If there were 2000 keys with numeral one, then numeral two's buffer would start at element 3001. Combining this with the offsets, we get exact element positions for the keys of each node. Continuing the example, if the numeral-one offset for node 3 was 225, then the overall starting position for node 4's numeral-one keys would be $1001 + 225 = 1226$. The second numeral-one key at node 4 would then go into element 1227, and so on.

The large array is, of course, mapped across all of the processors. Thus, each element is really a node-index pair that specifies the node and index within that node's local array that corresponds to the element. Fortunately, we can use a simple abstraction to convert element numbers into node-index pairs: in particular, we use a fixed-sized array on each processor of size k elements and the following two equations:

$$\text{node}(i) \equiv \left\lfloor \frac{i}{k} \right\rfloor \qquad \text{index}(i) \equiv i \bmod k. \tag{34}$$

After we have the starting positions for all of the numerals, we are ready to rearrange the data. For each node and each numeral, we maintain the element position for the next key with that

1. In a *scan* operation, each node i contributes an element x_i and receives the value $x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$, where "$\oplus$" denotes an associative operator, in this case addition. Thus, each node gets the sum of the values to its left, with node 0 receiving 0. A *reduction* is similar, except that each node gets the total sum for all nodes. In a *vector scan*, each node contributes a vector and the result is the vector of the element-wise scans. Section 6.1.1 and Appendix A cover Strata's scan and reduction operations.

numeral. We make one pass through the data, sending the data according to the element number for its numeral, and then incrementing the element number. Because we have preallocated all of the target array space precisely, the keys map one-to-one with the target array elements.

We repeat this process for each digit. After all of the digits have been sorted we are done. We should note that we actually use two large arrays and alternate between them: the source for one digit becomes the target for the next digit.

Radix sort has a several interesting properties. First, it is stable, since the keys on a node with the same numeral maintain their relative position after the move. Second, the running time is independent of the keys: the distribution has no impact on the running time. Put another way, radix sort essentially always rearranges the data completely for each digit. For example, even sorted data would be rearranged many times, since the first step is to sort according to the least-significant digit. Third, there really is no "local sorting" phase, as normally expected for a parallel sorting algorithm: sorting occurs through the global counting operations and then the following global rearrangement of data.

4.5.2 Sample Sort

The primary draw back of radix sort is that it rearranges all of the data for every digit. Thus, a single key moves through a sequence of processors as it is sorted, eventually winding up at the correct processor. Ideally, we would just move the key directly to its final destination. This is the goal of sample sort, which uses sampling to guess the final destinations for each key. Equivalently, sample sort guesses the boundary keys for each node. Reif and Valiant developed the first sampling sort, called *flashsort*, in 1983 [RV87]. Another early version was implemented by Huang and Chow [HC83].

For n keys and p processors we get the following basic algorithm:

1. Find $p - 1$ keys, called *splitters*, that partition the key space into p buckets,
2. Rearrange the data into the p buckets, which correspond to the p processors,
3. Sort the keys within each bucket.

Unfortunately, it is possible for the splitters to be far enough off that one of the processors is the target for more keys than it can physically hold due to memory constraints. Furthermore, the wider the variance in keys per node, the longer the final local sorting phase will take, since the time of the phase is limited by the processor with the most keys.

Fortunately, there is a nice solution to both of these problems: we can use s buckets per processor, where s is called the *oversampling ratio*. For example, if $s = 10$ there is still a chance that one of the buckets will have many more keys than the average, but it very unlikely that all ten of the buckets on one processor will contain many more keys than the average. The expected number of keys per node is $\frac{n}{p}$; if we call the maximum number of keys per node L, then the we can define the work ratio, α, as:

```
Pick $s$ keys at random on each node, for a total of $ps$ keys
Sort the $ps$ keys using parallel radix sort
Send every $s^{th}$ key to node 0; these are the splitters
Node 0 broadcasts the splitters to all nodes
For each of the $\frac{n}{p}$ keys (on each node) do:
      Determine the target node using binary search
      Group the keys into buffers by destination processor
      When a buffer becomes full, send it to its node
end
Transfer all non-empty buffers
Block until all transfers are complete
      [by counting outstanding transfers]
Repeat algorithm for load balancing if needed
      [discussed in text]
Sort data locally to complete the sort
```

Figure 4-12: Pseudo-code for Sample Sort

$$\alpha \equiv \frac{L}{(n/p)}. \quad (35)$$

Blelloch *et al.* [BLM+92] prove the following bound on L:

$$\Pr\left[\frac{L}{(n/p)} > \alpha\right] \le n(e)^{-\frac{\alpha s}{2}\left(1-\frac{1}{\alpha}\right)^2}. \quad (36)$$

This inequality bounds the probability that the worst processor has more than α times as many keys as the average. For example, with $s = 64$ and $n = 10^6$ keys, the probability that the worst processor has more than $\alpha = 2.5$ times as many keys as the average is less than 10^{-6}. In practice, oversampling ratios of 32 or 64 ensure even load balancing of the keys, essentially always resulting in $\alpha < 2$.

Figure 4-12 shows the pseudo-code for sample sort with oversampling. There are many design options for this algorithm. One of them is how to find the p splitters, which requires sorting ps keys. Since we are implementing a parallel sorting algorithm, recursion is one logical option. However, it is much simpler and faster to use the parallel radix sort that we already have. Although it potentially slower than sample sort, we are only sorting a small number of keys and the simplicity and small constants of radix sort turn out be a clear win.

Another decision is how to move the data to the target nodes. We allocate a fixed-size block-transfer buffer for each target on each processor, and then move the data into these buffers as we determine the appropriate node. If a buffer becomes full, we transfer it so that we can reuse the space and continue. At the end we have a series of partially full buffers, which we then transfer using a version of Strata's asynchronous block-transfer engine, which is discussed in Section 7.4.

There is some small chance that a node will have insufficient memory for the incoming keys. We assume space for $\frac{2n}{p}$ keys plus the buffer storage. When there is insufficient space for an incoming buffer, we forward it to a random node; we know that there is sufficient space somewhere. In the case, we simply restart the algorithm after the transfer phase, which is not as bad as it sounds for several reasons. First, we oversample by a factor of 64, so the chance that there is insufficient storage is near zero. Second, when we repeat the algorithm we get new splitters that are completely independent, so that we expect the second iteration to succeed even though the first one failed. If the probability of failure for one iteration is 10^{-6}, then the chance that k iterations are insufficient is 10^{-6k}. Third, since the distribution was mostly correct, most of the data is already on the correct processor and need not be transferred. Thus, although we expect the algorithm to complete in one iteration, it will actually iterate as needed to find a suitable set of splitters.

Given that sample sort almost always rearranges the data only once, it should beat radix sort for large numbers of keys per processor or for platforms with expensive communication. However, it is more complicated than radix sort, requires more memory, and has substantially more set-up work to perform. Thus, we expect radix sort to do well on smaller problems and perhaps on platforms with high-performance communication. Furthermore, the time for radix sort depends linearly on the width of the keys, while sample sort is essentially independent of the size of the key space.

4.5.3 The Sorting Module

The sorting module sorts integer keys of widths from 4 to 32 bits. It assumes each node contributes a fixed-size buffer that is at most half full of keys; after the sort, the buffers contain each node's chunk of the sorted keys, with keys strictly ordered by processor number. The module also returns a list of $p - 1$ keys that can be used to binary search for the node containing a given key.

The module has two parameters: the number of keys per node and the width of a key in bits. We select among four algorithms:

1. Sample Sort
2. Radix Sort with 4 bits per digit, radix = 16
3. Radix Sort with 10 bits per digit, radix = 1024
4. Radix Sort with 14 bits per digit, radix = 16384

The platforms remain the same: the CM-5 and simulated versions of Alewife, Paragon, and an ATM-based network of workstations.

4.5.4 Results

We start by looking at the relative performance on a 64-node CM-5 with 28-bit keys. Figure 4-13 shows the performance of the four algorithms. For this module, the slopes are always linear in the number of keys per node for a particular key width and platform. This means that each algo-

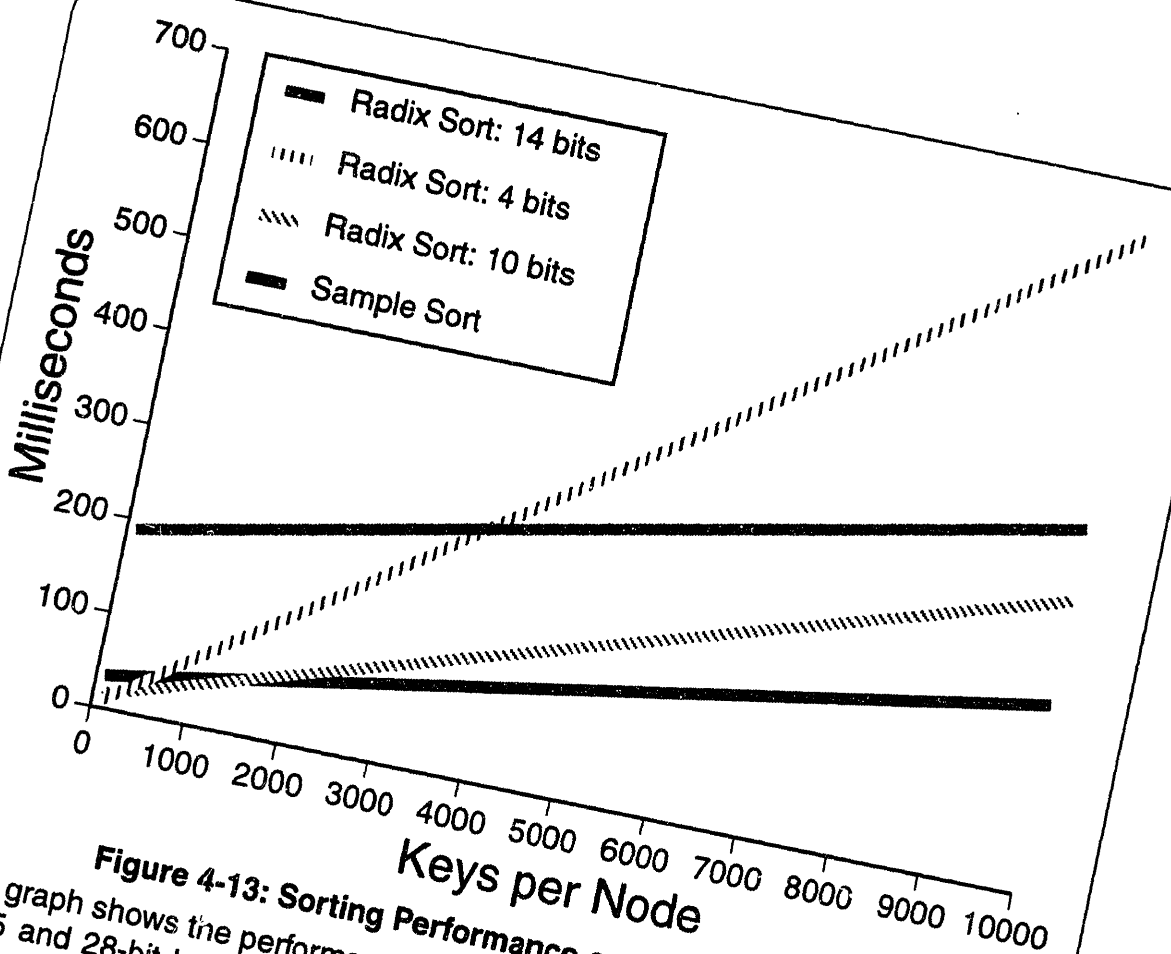


Figure 4-13: Sorting Performance on a 64-Node CM-5
This graph shows the performance of the four sorting algorithms for a 64-node CM-5 and 28-bit keys. Sample sort is the best for large problems, while the small-radix radix sorts work best for small problems.

rithm can only win for a single contiguous region of keys per node, and that the prediction graph can have at most four regions along the keys-per-node dimension for a given key width and platform.

Figure 4-14 shows the prediction graph for the 64-node CM-5. As expected, the small-radix radix sorts work best for small problems, while sample sort is the best asymptotically. Radix sort with a 14-bit radix never wins: although it is the best version of radix sort for larger problems, sample sort is even better for those same inputs.

Several bars contain black bars. These bars mark the actual crossover point for the corresponding region boundary. To clarify the predicted crossovers, the bars have been omitted if the actual crossover point is within 50 keys per node of the prediction, which is one-fortieth of an inch graphically.

Table 4-4 gives the predicted and actual regions for all of the platforms, along with a summary of the prediction errors. These values are for 64-node machines. For each platform, we looked at three different key widths: 32 bits, 16 bits, and 8 or 24 bits, whichever seemed more interesting. Given a platform and key width, we used the models to predict the appropriate range

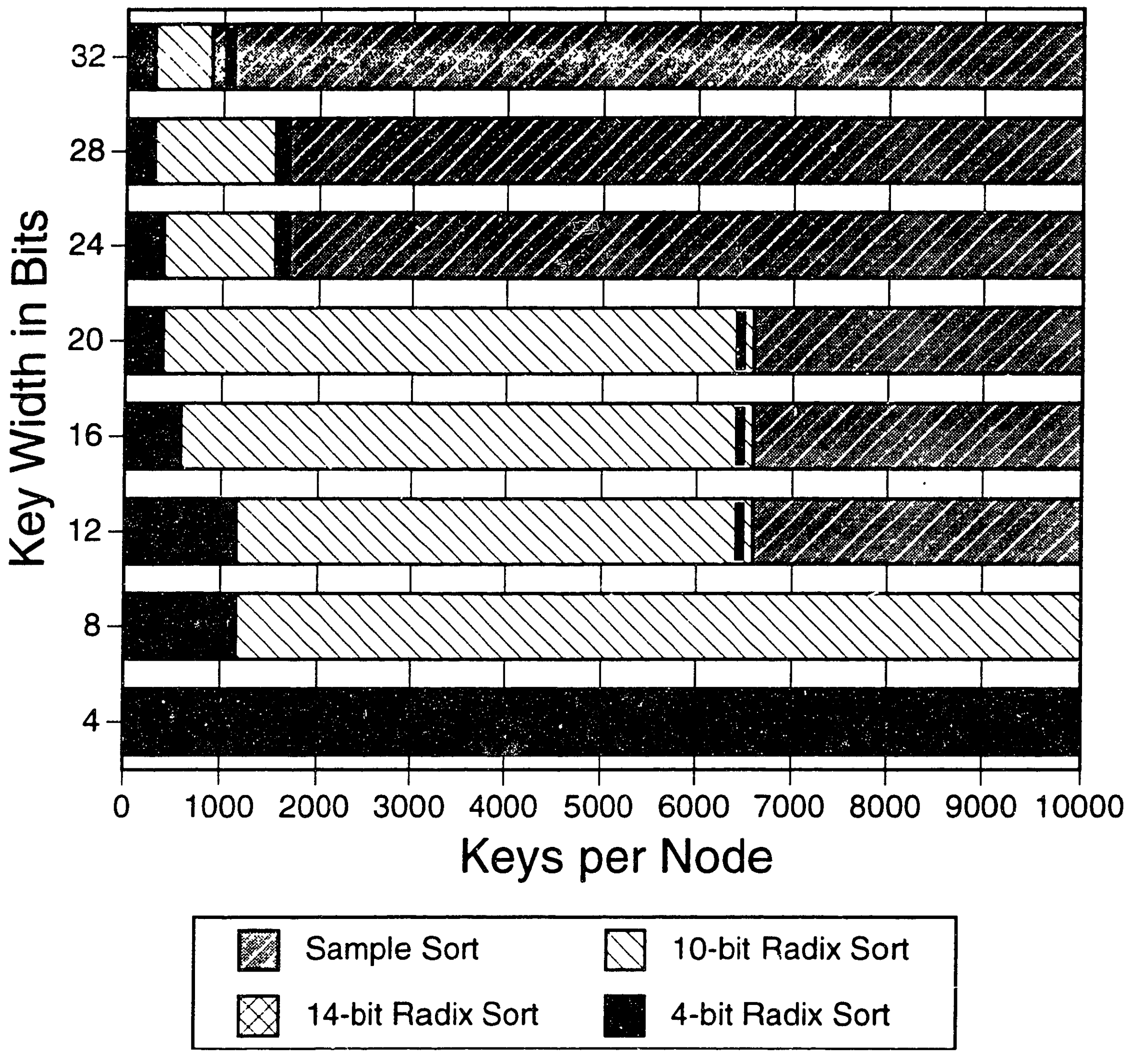


Figure 4-14: Predicted Regions for the CM-5

This plot shows the best algorithm for a given key width and keys per node. This plot is for a 64-node CM-5; the 32- and 128-node versions look nearly identical. These regions are the predictions: the black bars denote the actual crossover point. The bars are omitted for clarity if the predicted crossover is within 0.5% of the actual crossover point. Thus, the models accurately predict the best algorithm. Note that 14-bit Radix Sort never wins.

of problem sizes. These are shown as the "predicted" intervals. We then compared the algorithms empirically around the predicted crossover points to determine the actual crossover points; the actual ranges are shown as the "actual" interval. The appearance of a "—" indicates that the corresponding algorithm was never the best choice for that platform and key width. Given the predicted and actual intervals, we can compute the number of different keys-per-node values for which the prediction was wrong. For example, if a crossover point was off by ten keys, then we

Platform	Key Width	4-bit Radix	10-bit Radix	Sample Sort	Error Ratio
Alewife: predicted	16	1–2584	2585–	—	$\frac{20}{10000}$
Alewife: actual		1–2604	2605–	—	
Alewife: predicted	24	1–1295	—	1296–	$\frac{7}{10000}$
Alewife: actual		1–1288	—	1289–	
Alewife: predicted	32	1–817	—	818–	$\frac{2}{10000}$
Alewife: actual		1–819	—	820–	
CM-5: predicted	16	1–580	581–6597	6598–	$\frac{49}{10000}$
CM-5: actual		1–583	584–6551	6552–	
CM-5: predicted	24	1–386	387–1551	1552–	$\frac{61}{10000}$
CM-5: actual		1–391	392–1607	1608–	
CM-5: predicted	32	1–290	291–879	880–	$\frac{5}{10000}$
CM-5: actual		1–291	292–875	876–	
Paragon: predicted	8	1–661	662–	—	$\frac{31}{10000}$
Paragon: actual		1–692	693–	—	
Paragon: predicted	16	1–553	—	554–	$\frac{6}{10000}$
Paragon: actual		1–547	—	548–	
Paragon: predicted	32	1–215	—	216–	$\frac{0}{10000}$
Paragon: actual		1–215	—	216–	
ATM: predicted	8	1–383	384–	—	$\frac{8}{10000}$
ATM: actual		1–391	392–	—	
ATM: predicted	16	1–312	—	313–	$\frac{6}{10000}$
ATM: actual		1-318	—	319–	
ATM: predicted	32	1–126	—	127–	$\frac{1}{10000}$
ATM: actual		1-127	—	128–	

Table 4-4: Prediction Accuracy for Sorting

This table gives the percentage of incorrect predictions for keys per node in the range of 1 to 10,000. The overall mean accuracy is 99.84%.

count that error as ten incorrect predictions. We take the total number of incorrect predictions and divide by 10000, which is our nominal range of interest, to get the "Error Ratio" column. We leave the ratio as a fraction because the denominator is rather arbitrary: higher values would imply better accuracy. Nonetheless, the number of prediction errors is much smaller than the range of interest in practice, which covers at least a few thousand keys per node.

In general, radix sort does much better on the CM-5 than on the other platforms. This is primarily due to two factors: the hardware support for vector scan operations, and the absence of an integer multiplication instruction in the CM-5's Sparc instruction set, which slows down the random-number generation required by sample sort. The three simulated architectures use a combining tree for vector-scan operations, and use the MIPS R3000 instruction set, which provides a multiplication instruction.

Sample sort almost always wins eventually. That is, usually there is some size above which sample sort is always the best choice. The exception to this rule is that radix sort wins if it only requires one iteration. For example, with a key width of 8, the 10-bit radix sort requires only one pass. Sample sort also requires only one pass, but it has much higher overhead. The more interesting case is the Alewife platform with 16-bit keys, in which 10-bit radix sort requires two passes to sample sort's one. However, the communication performance of the Alewife is fast enough that the cost of a second pass for radix sort is less than the additional overhead for sample sort, which involves a binary search for each key and a local sort at the end. In general, radix sort is competitive only for small problems or small key widths.

For the stencil module, we showed that the penalty for prediction errors was small because the errors only occur when there are at least two viable options. This applies even more to this module. In fact, it is nearly impossible to measure a penalty that is statistically significant. This is due to the fact that the performance of sample sort varies depending on the random-number generator. Thus, the crossover points from radix sort to sample sort are always nebulous: depending on the choice of random numbers, values near the crossover point could go either way. As we move away from the crossover point, the difference in performance starts to dominate the white noise from the random-number generator and there would be a measurable penalty. But there are no errors away from the crossover points! Overall, we expect a small penalty, but we have not determined it precisely. Based on 95% confidence intervals for 12 samples, the penalty is less than 0.5% for 32 different randomly selected errors, which at least confirms our expectation that the penalty for errors is very small.[1]

Although we can explain all of the effects in retrospect, it is critical to emphasize that the models capture this information implicitly: there is no conceptual understanding involved other than the designer's choice of terms. Thus, the models allows us to leverage our high-level understanding of each algorithm into a complete understanding of exactly when to use each one on each platform.

1. Actually, I just picked 32 error points with the intent to cover all of the crossover intervals from the table at least once.

```
float Radix(float keys, float bpd, float width) { /* MRE: 1.96% */
      float iter = ceiling(width/bdp);
      float bins = pow(2, bdp);
      return(502 + 7.18*iter + 2.64*bins*iter + 10.1*logP*iter
          +2.70*bins*iter*logP + 28.16*keys*iter);
}

float Sample(float keys) { /* MRE: 3.22% */
      return(16663 + Radix(64, 8, 32) + 0.51*keys*logP + 33.74*keys);
}
```

Paragon Models

```
float Radix(float keys, float bpd, float width) { /* MRE: 2.21% */
      float iter = ceiling(width/bdp);
      float bins = pow(2, bdp);
      return(502 + 7.18*iter + 2.64*bins*iter + 10.1*logP*iter
          +2.70*bins*iter*logP + 28.16*keys*iter);
}

float Sample(float keys) { /* MRE: 3.44% */
      return(16515 + Radix(64, 8, 32) + 0.51*keys*logP + 67.80*keys);
}
```

ATM Network Models

Figure 4-15: Sorting Models for Paragon and the ATM Network

4.5.5 Models

Figures 4-10 and 4-10 present the models for the sorting module for the four platforms. These models are slightly simpler than those for the stencil module, with the CM-5 model for radix sort standing out as exceptionally simple and accurate:

$$\text{RadixSort}(keys, bpd, width) = 11.41 \cdot bins + 9.92 \cdot iter \cdot keys + 77.36 \cdot logP \qquad (37)$$

where:

$$keys \equiv \text{Number of keys per node}$$
$$bpd \equiv \text{Radix width in bits (bits per digit)}$$
$$width \equiv \text{Key width in bits}$$
$$iter \equiv \left\lceil \frac{width}{bpd} \right\rceil \equiv \text{Number of iterations}$$
$$bins \equiv 2^{bpd} \equiv \text{Number of buckets (the radix)}$$
$$logP \equiv \text{Log base 2 of the number of processors}$$

```
float Radix(float keys, float bpd, float width) { /* MRE: 1.81% */
      float iter = ceiling(width/bdp);
      float bins = pow(2, bdp);
      return(481 + 5.45*iter + 2.31*bins*iter + 8.73*logP*iter
          +2.70*bins*iter*logP + 6.41*keys*iter);
}

float Sample(float keys) { /* MRE: 3.02% */
      return(11651 + Radix(64, 8, 32) + 0.51*keys*logP + 11.96*keys);
}
```

Alewife Models

```
float Radix(float keys, float bpd, float width) { /* MRE: 1.33% */
      float iter = ceiling(width/bdp);
      float bins = pow(2, bdp);
      return(11.41*bins + 9.92*iter*keys + 77.36*logP);
}

float Sample(float keys) { /* MRE: 2.86% */
      return(26344 + Radix(64, 8, 32) + 0.43*keys*logP + 14.21*keys);
}
```

CM-5 Models

Figure 4-16: Sorting Models for Alewife and the CM-5

The first term is the cost of the buckets, which require global vector scans and reductions. The second term is the cost of rearranging the data, which depends of the number of iterations and the number of keys per node. The last term gives the marginal cost for the machine size. Thus, this model captures the performance of radix sort on the CM-5 with only three terms and yet achieves excellent accuracy (MRE = 1.33%). The original list of terms included many other combinations that seemed important, such as the number of processors, but the toolkit threw them out as superfluous; this model's power and simplicity make it the best in the thesis.

In general, the sample sort models are less accurate than those for radix sort (and, in fact, for the rest of the models in this work). This makes sense given the variance in execution times due to the random-number generator: some runs are more load balanced than others.

4.6 Run-Time Model Evaluation

Although it has only been hinted at so far, the model-based decisions can be made at run-time as well as at compile time. Postponing the decision until run-time allows the system to exploit the exact parameters of the problem, such as the size of the virtual grid or the number of keys per node.

The auto-calibration toolkit makes run-time evaluation simple by generating C models that we can link in with the application. Thus, for run-time decisions the module contains all of the implementations, all of the models, and an initialization procedure that performs the evaluations and then executes the selected algorithm.

There are two drawbacks to run-time selection: the increase in code size and the overhead required to perform the selection. The code expansion is very application dependent, but is mitigated by the fact that all of the implementations share the Strata and library code. The expansion for the stencil module, which has five implementations, was 2.9 times larger than any single one.[1] The sorting module, which has three implementations, experienced an expansion of 210% relative to any of the radix sorts, and a 150% expansion relative to sample sort. These numbers measure the expansion on disk; the memory requirements at run-time should be significantly better, since all but one of the modules will never be executed and thus never loaded into physical memory.

The overhead for run-time selection is about two microseconds per implementation, although it again depends on the application and the platform. For the five-implementation stencil module, run-time selection took 10.2 microseconds on the CM-5 and 8.4 microseconds on the PROTEUS platforms. The three-implementation sorting module required 6.0 microseconds on the CM-5 and 4.6 microseconds on the PROTEUS platforms. These costs are trivial compared to the execution times, which range from milliseconds to minutes.

Thus, the costs of run-time selection are minimal. The code expansion is obviously fundamental to the ability to use different algorithms, but the actual expansion is surprisingly small due to the effect of shared code. The time overhead perhaps could be reduced, but is only a tiny fraction of the overall execution time and thus can be essentially ignored.

4.7 Conclusions

In this chapter, we looked at two high-level libraries with multiple implementations. The goal was to determine the feasibility of using calibrated statistical models to select the best implementation for the target platform and workload. In the stencil module, the key issue is selecting the best data layout, while the goal for the sorting module is to select among versions of radix sort and sample sort. There are several important conclusions from these studies:

- ❑ The performance of the stencil module strongly depends on the layout of the virtual grid onto physical processors. We looked at six implementations and investigated four layouts in detail: square blocks, vertical strips, uniprocessor, and shared-memory with square blocks. The square-block layout is the best overall, but the uniprocessor and vertical strips version also have regions in which they are the best choice. The shared-memory version never wins due to its inefficient communication. We also showed the models can choose the layout with the best aspect ratio for grids of uneven dimensions. The models capture the impact of aspect ratio implicitly, along with additional information

1. The sixth implementation was removed from the stencil module, since it never wins. Likewise, the 14-bit version of radix sort was removed from the sorting module.

about the exact value of eliminating entire communication directions as is done by the vertical strips version.

❑ The performance of the sorting algorithms depends on the size of the problem, the width of the keys, and the target platform. We looked at three versions of radix sort and an implementation of sample sort. The radix sorts work best for small problems or small key widths, while sample sort is usually better asymptotically for larger keys (but not larger problems for small key widths). The models are very simple, but still had excellent predictive power.

❑ The models for both modules were quite accurate: the selector picked the best implementation more than 99% of the time on all of the platforms. In the few cases in which a suboptimal implementation was selected, that implementation was nearly as good as the best choice: only a few percent slower on average for the stencils and nearly identical for sorting. We showed that the system only makes errors near the boundary of the performance surfaces, from which it follows that the penalty for these errors is small.

❑ In contrast, the benefit of picking the right implementation was often very significant: averaging 8-90% for stencils and often more than a factor of two for sorting.

❑ An important benefit of automatic selection is the tremendous simplification of the implementations that arose from the ability to ignore painful parts of the input range. By adding preconditions into the models, we ensure that the selector never picks inappropriate implementations. Thus, an immediate consequence of automatic selection is the ability to combine several simple algorithms that only implement part of the input range.

❑ The selection can be done at run time as well as compile time. The code expansion is small considering that all of the versions must be linked in, and the overhead of evaluating the models is insignificant compared to the execution time of the selected implementation. Run-time selection provides a module that achieves the performance of the best available implementation for the given workload. Unlike traditional hybrid algorithms, the selection mechanism is tuned to the current environment and can ported with only a simple recalibration of the models.

❑ In general, porting a high-level library requires much less work than porting an application. The implementations depend on only C and Strata, and thus are trivial to port once those two pieces are available. Unlike traditional applications, high-level libraries require little if any tuning for the new platform. The libraries are essentially self-tuning: they combine the flexibility of multiple implementations with an automatic method for selecting the best one in the new environment. As long as at least one implementation runs well on the new platform, the library as a whole achieves good performance. Applications that use the libraries are also more portable, because many of the performance-critical decisions are embedded within the libraries and thus can be changed without affecting the application source code.

4.8 Related Work

The closest related work is that of Alan Sussman [Sus91][Sus92]. Sussman's system chooses the best mapping of data from a fixed set of choices for compilation onto a linear array of processors. He shows that the choice of mapping greatly affects the performance of two image processing algorithms and that his system picks the best mapping by using execution models. He also shows a case in which a single program requires different mappings for different problem sizes. This work extends Sussman's work by combining execution models with parameterized modules (discussed in the next chapter), by looking at far more general topologies, and by addressing algorithm decisions in addition to partitioning decisions.

Another body of related work is the use of performance predictors for parallel programs. Predictors for Fortran include PTRAN [ABC+88] and Parafrase-2 [Pol+89]. PTRAN determines the maximum available parallelism for a semantically single-threaded Fortran program and can use this information to establish a lower bound on execution time for a given number of processors. Parafrase-2 extends this by breaking the execution time into categories based on various operations such as floating-point operations and memory accesses. It also tracks the predicted critical path as an estimate of execution time. Although these estimates are very rough they are helpful in avoiding "optimizations" that introduce more overhead than their predicted benefit.

There are many systems that use models for performance prediction. Cytron [Cyt85] looks at determining the optimal number of processors to use for a fork-join task graph such as those generated by a "doall" loop in Fortran. He also determines the minimum task size required for a given level of efficiency.

Chen, Choo and Li [CCL88] present a simple performance model based on the number of processors, the time for unit computation, the time for a unit-sized message send to a neighbor, and the time for a unit-sized message broadcast. Their optimization phase uses this model to evaluate the effects of a small set of transformations and data layouts.

Culler *et al.* [CKP+93] propose the *LogP* model as tool for algorithm development and analysis. The models proposed here subsume all three of these systems and implicitly handle many additional aspects of the target architecture, such as bisection-bandwidth limitations, OS overheads, and caching effects. In general, the models used in this work are more accurate and more robust, and also support automatic recalibration.

Gallivan *et al.* [GJMW91] investigate model-based Fortran performance prediction in two steps. First they build a database of "templates" that represent common loop patterns. A template captures the load-store behavior and the vector behavior of the Alliant FX/8 multiprocessor. The templates are simple models based on profiling code for a representative loop. The second step of the process is to map application loops onto those in the database, which is done through compiler analysis. There will be inaccuracies depending on how close a match the compiler can find. They only handle loops that avoid conditional statements, procedure calls, and nonlinear array indices. The biggest difference with this work is scale: we look at whole phases rather than just nested loops. Looking at phases allows us to avoid their restrictions because we amortize the effects of conditional statements over time, which gives us a reliable average execution time. We also automate the modeling, which allows portability.

4.8: Related Work

A. Dain Samples' doctoral dissertation [Sam91] presents a profile-driven optimizer for sequential programs. It selects among implementations of a sequential module based on profile information for each implementation. Our system generalizes this technique, applies it to parallel computing (where the costs vary much more), uses statistical models as the medium for feedback, and separates the modeling from the decision making, which extends and simplifies the use of the profiling information.

Some systems use programmer-supplied annotations to control the implementation of a module. The most famous of these is High Performance Fortran [HPF], which uses programmer directives to control the layout and partitioning of arrays. The Prelude language [WBC+91], on which the author worked, used annotations to control the migration of data and threads to improve performance. The annotations were always viewed as a short-term solution that allowed the exploration of mechanisms, but side-stepped the problem of automatic selection of the appropriate mechanism.

A different class of related work revolves around the modularity issues. *Multipol* is a library of distributed objects for multiprocessors [Yel92]. Multipol provides some of the modularity and code reuse properties achieved in this work, but does not address automatic selection of the best implementation or the parameter optimization investigated in the next chapter. Many languages provides substantial support for this kind of modularity, including Modula-3 [Nel91] and Standard ML [MTH90], but none address automatic selection of multiple implementations.

A. Dain Samples' doctoral dissertation [Sam91] presents a profile-driven optimizer for sequential programs. It selects among implementations of a sequential module based on profile information for each implementation. Our system generalizes this technique, applies it to parallel computing (where the costs vary much more), uses statistical models as the medium for feedback, and separates the modeling from the decision making, which extends and simplifies the use of the profiling information.

Some systems use programmer-supplied annotations to control the implementation of a module. The most famous of these is High Performance Fortran [HPF], which uses programmer directives to control the layout and partitioning of arrays. The Prelude language [WBC+91], on which the author worked, used annotations to control the migration of data and threads to improve performance. The annotations were always viewed as a short-term solution that allowed the exploration of mechanisms, but side-stepped the problem of automatic selection of the appropriate mechanism.

A different class of related work revolves around the modularity issues. *Multipol* is a library of distributed objects for multiprocessors [Yel92]. Multipol provides some of the modularity and code reuse properties achieved in this work, but does not address automatic selection of the best implementation or the parameter optimization investigated in the next chapter. Many languages provides substantial support for this kind of modularity, including Modula-3 [Nel91] and Standard ML [MTH90], but none address automatic selection of multiple implementations.

5

Automatic Parameter Optimization

In this chapter, we extend the usefulness of automatic algorithm selection by providing support for parameterized algorithms. In the last chapter, we chose from among a small fixed set of algorithms. For both of the libraries we examined, many design decisions were hidden from the selection process in order to keep the number of choices reasonable. In this chapter, we show how to optimize module parameters, which in turn allows us to fold entire families of algorithms into the selection process.

In particular, we look at two parameterized variations of algorithms used in the last chapter. First, we develop a parameterized version of the square-layout stencil code that uses two parameters to improve load balancing: one for the extra number of rows to give to nodes on the top and bottom boundaries, and one for the extra number of columns to give to nodes on the left and right boundaries. These processors perform less communication and thus sit idle part of the time unless we give them additional grid points.

The second parameterized algorithm is radix sort with a parameter for the radix. As you may recall, a small radix requires several complete rearrangements of the keys, but has low overhead for each such iteration. A large radix requires fewer passes, but has significantly more work per iteration. We show that the optimal radix, hereafter called the *digit width*, depends on both the platform and the number of keys being sorted.

We show how the models can be used to find the optimal setting for each parameter for the target platform and workload. In *every single test* of this technique, the parameters were in fact set optimally. The reasons for this success rate include the accuracy of the models, the wide performance gaps among the parameter values, and the discrete nature of the parameters. For example, to find the optimal value for an integer parameter, the models need only be accurate enough so that rounding produces the correct integer.

The models thus provide a way to determine the optimal value of a parameter even though that value may depend on both the target platform and workload. The optimization can be done efficiently at compile time or run time. The ability to set parameters optimally and automatically greatly increases the prospects for portability with high-performance: the library designer determines which parameters matter, and the optimizer sets each parameter optimally, regardless of the underlying costs of the platform.

5.1 Parameter Optimization

The key novel idea in this chapter to is combine two powerful techniques: automatically calibrated statistical models and numerical root finding. Given models that accurately predict the cost of the code for a given parameter, we can use root finding to give us the optimal value. We do this in two different ways depending on the nature of the models. For both methods, the designer specifies the range of interest of the parameter.

First, if we can set up the problem as the equality of two models, then we can build a new model that computes the difference of the models. The root of the new model is the equality point of the original models. For example, in the stencil application the optimal value for the number of extra rows to give boundary nodes occurs when the cost of the extra rows matches the cost of the missing communication (because the processor is on the boundary). We build statistical models for the two costs (communication and computation) in terms of the parameter, and then find the root of the difference between the models to get the optimal parameter value. The root finder is:

```
int IntegerRoot(int bottom, int top, float f(int param))
```

This procedure finds the root of `f` in the interval [`bottom`, `top`]. If there is no root in the given range, then the root finder returns `bottom-1` if the curve is strictly negative, and `top+1` if the curve is strictly positive. If there is more than one root, the root finder returns one of them.

In the second method, we assume that the model has a single minimum and use a technique very similar to root finding to locate the minimum. Essentially, the algorithm is to use binary search, using the numerical derivative at each point to indicate in which half the minimum resides. This is equivalent to find the root of the symbolic derivative, but does not require any understanding of the model, other than the knowledge that the curve is smooth, continuous, and has exactly one minimum. The minimum finder is:

```
int IntegerMinimum(int bottom, int top, float f(int param))
```

This procedure find the minimum of `f` in the interval [`bottom`, `top`]. There is always at least one minimum, although it may be one of the endpoints. If there are multiple minima, the root finder returns one of them.

The overall methodology, then, is to build a composite model, $f(x)$, that either has a root or a minimum, and then to use one of the above procedures to find the optimal value of x. This can be done either at compile time, in which case we use perl versions of these functions, or at run time, in which case we use the C versions.

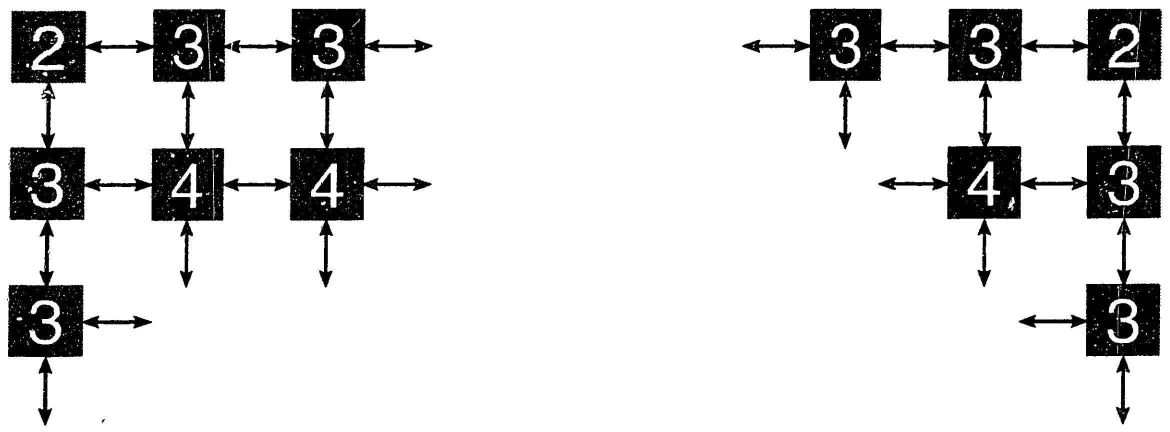


Figure 5-1: Communication Directions for Stencil Nodes
Boundary processors have fewer neighbors to communicate with than nodes in the center. The corner processors only perform two block transfers per iteration, compared with four for the internal nodes. The reduced communication means that these nodes sit idle part of the time. We can eliminate this waste by giving boundary processors additional points from the virtual grid. The key question becomes how many extra rows or columns should the boundary processors get to make up for their reduced communication?

5.2 Stencils: Extra Rows and Columns

A simple but useful optimization for stencil programs is to give the boundary processors extra work to do to make up for their reduced communication. Figure 5-1 shows the number of communication directions for different nodes in the square-block gird layout. For example, processors on the left edge do not have any left neighbors and thus communicate only three times, except for the corner processors, which have only two neighbors.

Nodes with fewer directions sit idle part of the time, since the middle nodes have the same amount of computation to do, but have more communication. Thus, we can give the boundary processors more grid points, and the middle nodes correspondingly less. This provides perfect load balancing and thus improves performance. We should point out however, that the benefit of this optimization is small for large machines, since there are relatively few boundary nodes. Nonetheless, this optimization never hurts and often helps; it is also a good example of the methodology.

5.2.1 The Composite Model

To build our composite model, we combine a model for the cost of the missing communication with a model for the cost of the additional grid points. The model for the missing communication is:

$$\text{Communication}\,(width) \;=\; StencilHeight \cdot \text{BlockTransfer}\left(\frac{8 \cdot width}{2\,(8)}\right) \tag{38}$$

where:

$$width \equiv \text{Width of the virtual grid}$$
$$StencilHeight \equiv \text{Number of rows along the top and bottom boundary that must be transmitted vertically (normally 1)}$$
$$\text{BlockTransfer} \equiv \text{The cost of sending a block transfer}$$

The block-transfer size is multiplied by 8 to convert doubles to bytes, divided by two because of red-black ordering, and divided by 8 because the width is divided among 8 nodes.

This is a qualitatively different model than we have used before. The statistical fitting is hidden within the BlockTransfer function, which is one of the Strata models. We could fit a model directly for the communication cost, but this version, called a *factored* model, has several advantages. First, the model never has to be recalibrated, since the Strata models are already recalibrated for each target architecture. Second, factored models allow the designer to make application predictions for a new platform by only estimating the low-level Strata models, which is considerably easier. Thus, a factored model allows the designer to leverage the Strata models to give reasonably accurate application predictions. Factored models are a recent development in this work and are presented here because they were actually used for the parameter optimization, which has more tolerance for the lower accuracy of a factored model. Chapter 8 covers factored models in more detail.

The model for the marginal communication cost is a regular model, in fact, it uses the models we already have:

$$\text{Computation}\,(w, h, x) \;=\; \frac{\text{Square}\,(w, h + 8x, 1000) - \text{Square}\,(w, h, 1000)}{1000} \tag{39}$$

where:

$$w, h \equiv \text{Width and height of the virtual grid}$$
$$x \equiv \text{The number of extra rows}$$
$$\text{Square} \equiv \text{The model for the square-block layout}$$

In this model, we exploit the fact that the marginal cost for x extra rows per node is the same as the marginal cost for $8x$ extra rows for the whole problem, since there are 8 processors along the vertical dimension. We compute the cost for 1000 iterations and then divide by 1000 to get the marginal cost per iteration; this effectively throws out initialization costs. Also, note that the marginal cost depends on the existing size of the virtual grid: an additional long row takes longer than an additional short row. The Square model already captures this behavior, so it is logical to use it in this context.

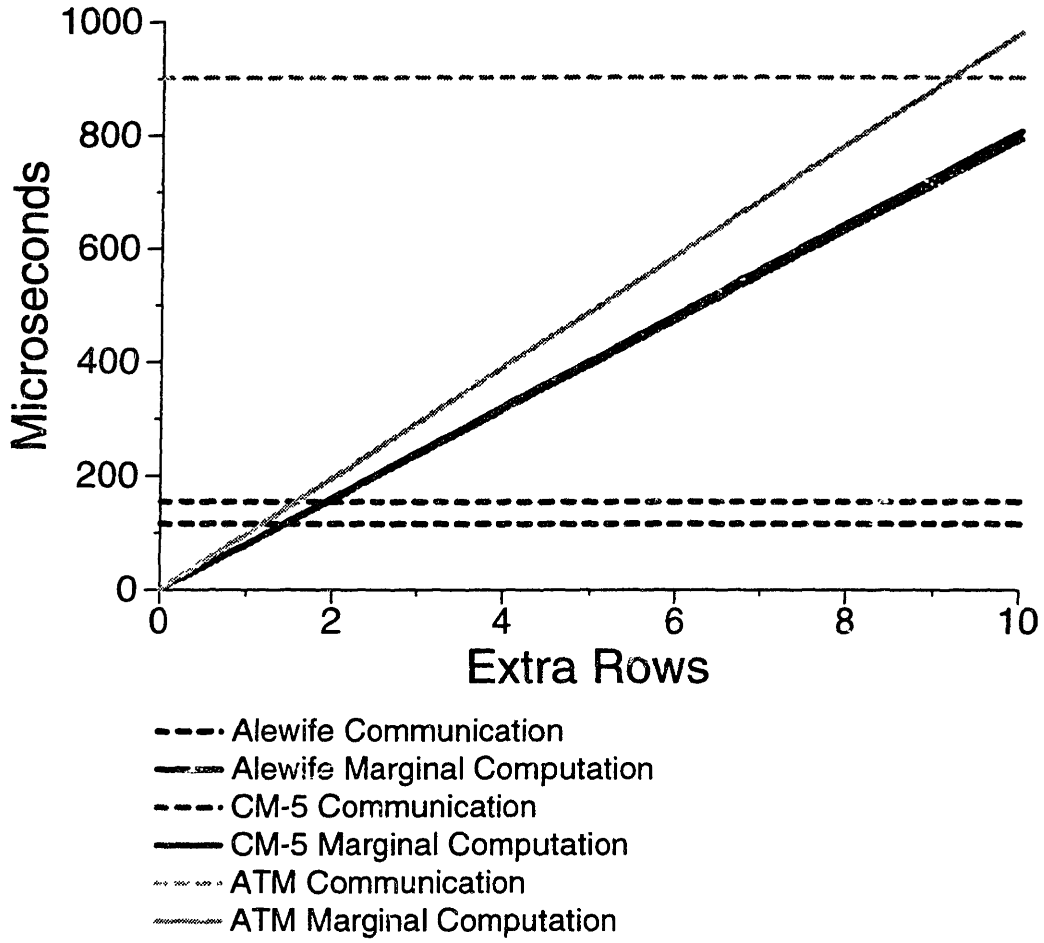


Figure 5-2: Extra Rows for Alewife and ATM

This graph shows the marginal computation cost for each extra row along with the fixed missing communication cost. The optimal value is the intersection of the two curves. Because of the differences in the cost of communication, the ATM requires 9 extra rows for optimal load balancing, while the Alewife and CM-5 only require one extra row.

Figure 5-2 shows the two models for the Alewife and ATM platforms; the CM-5 and Paragon look similar to the Alewife curves. Given our two submodels, we define a composite model as the difference between them:

$$f(x) = \text{Communication}(width) - \text{Computation}(width, height, x) \quad (40)$$

We give this function to the IntegerRoot procedure to calculate the optimal number of extra rows. A similar function is used for the optimal number of extra columns.

Platform, Input 64 Processors	Rows		Columns		Benefit	
	Pred	Opt	Pred	Opt	Pred	Actual
Alewife, 200x200	1	✓	1	✓	1.6%	1.4%
CM5, 200x200	2	✓	1	✓	3.1%	2.5%
ATM, 50x50	9	✓	2	✓	8.9%	8.1%
Paragon, 200x200	3	✓	1	✓	3.3%	2.6%

Table 5-1: Predicted Extra Rows and Columns and the Benefit
This table shows the predicted number of extra row and columns required to provide load balancing. All of the predictions were correct (when rounded down). The models also estimate the benefit of the optimization, usually quite accurately.

5.2.2 Results

Table 5-1 shows the predictions for the optimal number of extra rows and columns. Overall, we tried 100 different configurations of platforms and problem sizes: in all 100 trials, the predicted value was in fact optimal.

Furthermore, we can also use the models to predict the benefit of the optimization. In particular, if we assume that the extra rows and columns simply fill up idle time on the boundary processors, then the performance of the optimized version is identical to the performance of a problem with that many fewer rows and columns. For example, with r extra rows and c extra columns we get:

$$\text{OptimizedSquare}\,(width, height, iter) \approx \text{Square}\,(width - 2c, height - 2r, iter) \qquad (41)$$

Table 5-1 gives the predicted and actual benefit of the optimization. The predicted value is uniformly high, but not by much. The error is due to the fact that the optimal number of rows and columns would ideally be fractional; for example, it might be the case that the communication costs two-thirds of a row. Of course, the implementation requires an integer number of additional rows, and thus the cost of a whole row can not always be hidden in the idle time. Overall, the predicted benefit is more than sufficiently accurate to determine if the optimization is worthwhile. Finally, we note that as expected, the benefit of the optimization is relatively small, and has less impact on bigger problems, since a larger problem divides the benefit, which is essentially fixed for the platform, by a larger amount of work.

Thus, the models are able to determine reliably the optimal value for these parameters, even though they depend on the platform, and to a lesser degree, on the problem size. We can also use our existing models to predict the benefit of the optimization.

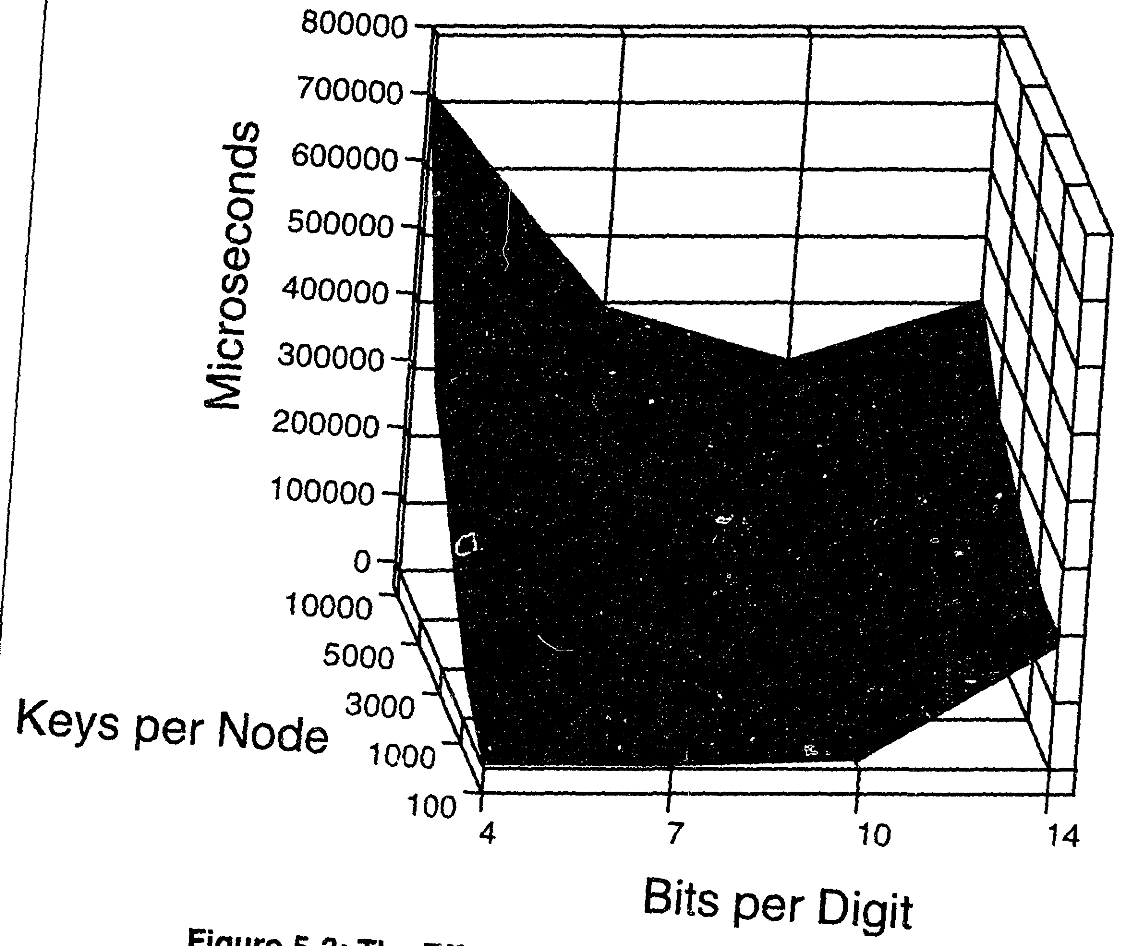


Figure 5-3: The Effect of Digit Size on Radix Sort
This graph plots the performance of radix sort on a 64-node CM-5 and 28-bit keys, versus the width of the digit in bits and the number of keys per node. The optimal digit size depends on both parameters. For less than 1000 keys per node, the optimal width is 4 bits, around 1000 keys it is 7 bits, and above 1000 keys it is 10 bits. For large enough sorts, 14-bit digits would be optimal.

5.3 Radix Sort: Optimal Digit Size

In our second example, we attempt to pick the optimal radix for radix sort. The best radix depends on the platform, the problem size, and the width of the keys. Figure 5-3 shows the effect of the digit size and problem size on the performance of radix sort for a 64-node CM-5 with 28-bit keys. In the last chapter, we used algorithm selection to pick the best radix. This made it difficult to provide a wide range of radix values, since each one had to be a separate implementation. By using parameter optimization, we can build one implementation that covers all of the different radix values.

```
float Radix(float keys, float bpd, float width) { /* MRE: 1.33% */
      float iter = ceiling(width/bdp);
      float bins = pow(2, bdp);
      return(11.41*bins + 9.92*iter*keys + 77.36*logP);
}
```

Figure 5-4: CM-5 Model for Radix Sort

5.3.1 The Composite Model

In fact, our existing models for radix sort already take the digit size as a parameter. Figure 5-4 repeats the CM-5 radix sort model for reference. Thus, we can build a composite model quite trivially:

$$f(x) = \text{Radix}(keys, x, key\_width) \quad (42)$$

where:

$$keys \equiv \text{The problem size in keys per node}$$
$$key\_width \equiv \text{The width of the keys}$$

To find the optimal radix, we simply invoke `IntegerMinima` with *f* and a reasonable interval, such as [1, 16].

Unfortunately, this does not quite work: the specified f may have multiple minima, in which case we may get only a local minimum rather than the best choice. Figure 5-5 shows the graph of $f(x)$ versus x for a 64-node CM-5 with 28-bit keys and 1000 keys per node. The drop from 9 to 10 occurs because a 9-bit radix requires four iterations, while a 10-bit radix needs only three. Fortunately, we can build a better composite model that avoids this problem.

The key idea is to build a model that only looks at the minimum radix size for a given number of iterations. Thus, we parameterize the composite model in terms of iterations, from which we can compute the best radix. In particular, given keys of width *key_width* and *i* iterations, the best digit width is:

$$\left\lceil \frac{key\_width}{i} \right\rceil . \quad (43)$$

This leads to the following composite model:

$$g(i) = \text{Radix}\left(keys, \left\lceil \frac{key\_width}{i} \right\rceil, key\_width\right). \quad (44)$$

The new model only visits the best size for each iteration, which provides a function with only one minimum. The viable sizes are circled in Figure 5-5; connecting the only the circled nodes does in fact provide a curve with exactly one minimum at $i = 4$, which corresponds to a digit width of 7 bits.[1]

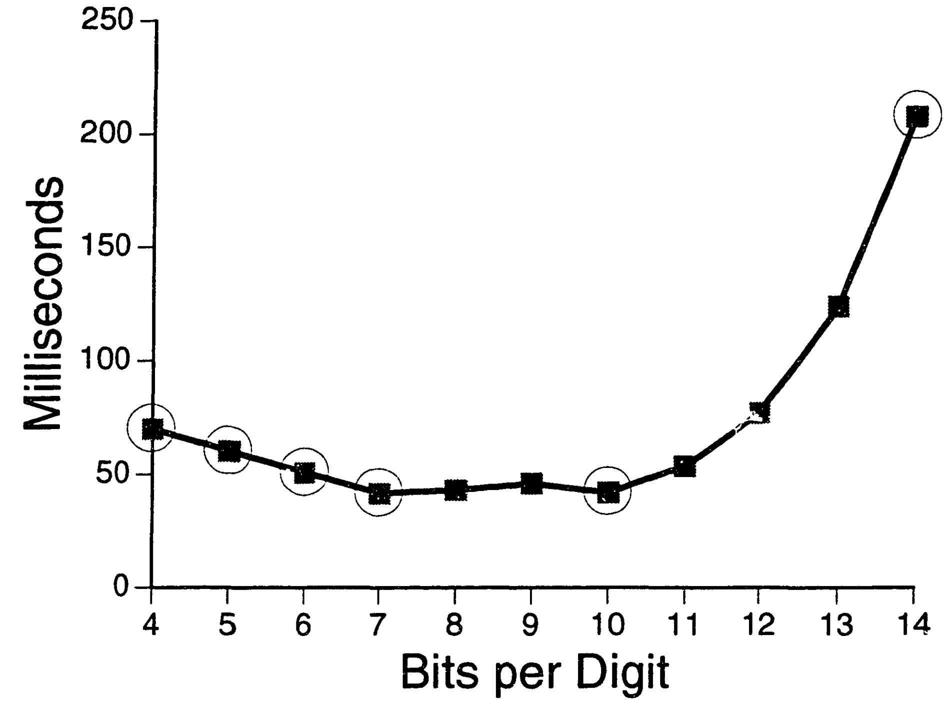


Figure 5-5: Multiple Minima in the Composite Model
This graph plots the proposed composite model from equation (42) against the parameter (bits per digit). The graph has multiple minima and thus could lead to suboptimal values. Only the circled nodes are considered by the revised model of equation (44), which only has one minimum.

5.3.2 Results

As with the stencil example, the predicted optimal radix was correct in all trials, in this case 40 trials. We checked optimality by selecting values near the predicted value and confirming that there was indeed a local minimum. We also checked values that were not considered by the revised composite model.

It is difficult to access the benefit of this optimization. The overall best choice is probably 8-bit digits, but it is not obvious. In any case, any single choice will be very bad for some inputs. For example, with a key width of 9, 9-bit digits can perform up to twice as well as 8-bit digits. Similarly, with a key width of 4, 4-bit digits on small problems can outperform 8-bit digits by more than a factor of three.

Although the optimal value is important for good performance, it is very difficult to determine manually. The platform, problem size, and key width all have substantial impact on the best digit

1. It is difficult to tell visually, but the 10-bit radix performs 0.31 milliseconds slower than the 7-bit version.

width, yet it is not obvious how to combine these influences to determine the optimal value. Optimizing any single one alone leads to situations with poor performance. The beauty of model-based parameter optimization is the ease with which it perfectly combines these influences: the composite model precisely captures the importance of each factor.

Given this family of radix sorts that use the optimal digit width, we can now build a better sorting module. In particular, to perform algorithm selection, we first determine the optimal digit width. We then select between radix sort with the optimal radix and sample sort. This results in a module with fewer implementations and better overall performance. It also increases the benefit of automatic selection, since there are now more situations in which radix sort is the best algorithm.

5.4 Run-Time Optimization

Parameter optimization can be also be done at compile time or run time, but if done at compile time, some of the relevant inputs, such as the number of keys per node for radix sort, might have to be filled in with generic values. Furthermore, unlike algorithm selection, run-time parameter optimization avoids code expansion. Thus, run-time parameter optimization should be a win most of the time.

The potential drawback is the overhead of determining the optimal value. For the stencil module, each of the two optimizations took 8 iterations inside the root finder and a total time of 75 microseconds on the CM-5 for the interval [0, 100]. Using a more reasonable interval of [0,20] took 5 iterations and a total time of 51 microseconds. The times for radix sort were similar, since the dominant cost is the number of iterations of the root finder. Thus, the overall cost is about 10 microseconds per iteration, which again is insignificant compared to the execution times of milliseconds to minutes. (Just *loading* the program takes milliseconds.)

5.5 Conclusions

This chapter used the models for parameter optimization, which is theoretically more difficult than algorithm selection. The results from this chapter reveal that in some ways parameter optimization is easier: in particular, there is more room for error, since small changes in the parameter tend to lead to rather large changes in performance. Put another way, it is harder for two points on the same curve to be close for a discrete parameter, than it is for two different curves to be close. Thus, although there were a few errors in prediction for algorithm selection, we found no errors in the prediction of the optimal parameter value. We looked at 80 trials of each of the two stencil parameters, and 40 trials of the digit-width parameter for radix sort.

Model-based parameter optimization provides a form of adaptive algorithm that remains portable. In fact, one way to view this technique is as a method to turn algorithms with parameters that are difficult to set well into adaptive algorithms that compute the best value. As with algorithm selection, this brings a new level of performance to portable applications: the key performance parameters are set optimally and automatically as the environment changes.

The structure of the root finders requires that the composite models used to determine the optimal value have a single root or minima, as appropriate. This led to complications for the radix sort model, but it was relatively simple to change the parameter from digit width to iterations, which solved the problem. It may be that other optimizations require a more sophisticated tool for finding the global minimum, but for these applications the single-minimum requirement was sufficient.

Run-time parameter optimization has low overhead, no code expansion, and can exploit workload information, such as the problem size, that is only available dynamically. Thus, unless all of the factors are known statically, it is better to postpone parameter optimization until run time.

Finally, parameter optimization can be combined with automatic algorithm selection to produce simple, more powerful modules, with better overall performance. The selection is done in two passes: first, we find the optimal parameter values for each of the parameterized implementations, and then we select among the optimized versions. A subtle caveat in this methodology is that run-time parameter optimization requires run-time algorithm selection, which leads to code expansion. This is not really a problem, but parameter optimization by itself avoids code expansion.

5.6 Related Work

The only real related work is the myriad of adaptive algorithms. These algorithms use feedback to adjust the key parameter dynamically, which although quite robust, may take some time to reach the optimal value. A good example is the "slow start" mechanism used in current TCP implementations [Jac88]. When a connection is established, the transmission rate is limited to something small to ensure success. The rate, which is the key parameter, is then slowly increased until the connection starts to lose packets, presumably because the rate exceeds that of either the receiver or one of the intermediate links. Once this crossover point has been found, which can take several seconds, the rate is reduced to a sustainable level. For short messages, which are becoming common with the popularity of the World Wide Web, the optimal value is never reached, and the connection achieves very low bandwidth.

In contrast, model-based parameter optimization always uses the optimal value for static evaluation. For run-time evaluation, there is some overhead to compute the value, but probably much less than any feedback-based adaptive algorithm, which fundamentally requires several iterations to converge. The disadvantage of model-based optimization is that it can be wrong, although for our applications it was always correct. The feedback-based algorithms are simple more robust: they can handle dynamic changes in the environment and unknown environments.

Model-based optimization in an unknown environment requires data collection and model building, which is just a fancy form of feedback. There could be some advantages to a hybrid algorithm: one could cache the models and then perform model building only for uncached environments. This would allow immediate use of the optimal parameter setting for cached environments. Caching the model rather than just the parameter value would allow us to adjust the parameter based on the workload. As a simple example, TCP could cache rates for sender-receiver pairs; since the rate is just a starting point for the feedback mechanism, the algorithm remains robust. This example only requires that we cache the rate, since the optimal rate is independent of the application traffic.

6

The Strata Run-Time System

Strata is a multi-layer communication package that was originally designed and implemented for the CM-5. The Strata run-time system is the core of that package; it contains support for communication, debugging and data collection. Strata has many goals, but the three most important are safe high-performance access to the hardware, predictability, and ease of development. The goal of Strata in this context is an efficient and predictable run-time system, upon which we can build portable, predictable high-performance applications. Many of its features resemble aspects of the PROTEUS simulator; in fact, the painful difference in development environments between PROTEUS and the CM-5 led to many of Strata's development features.

This chapter describes the novel aspects of Strata, its impact on high-level libraries, and its models, which were generated by the auto-calibration toolkit. Appendix A contains the Strata reference manual, which covers many important but mundane details and describes all of the Strata procedures. Finally, Robert Blumofe designed and implemented Strata's active-message layer.

6.1 The Strata Run-Time System

The Strata run-time system is part of a larger multi-layer communication package that is also called Strata. The run-time system provides four areas of functionality:

1. Synchronous Global Communication,
2. Active Messages,
3. Block Transfer, and
4. Support for Debugging and Monitoring.

The support for high-level communication, such as the cyclic shift (CSHIFT) communication pattern, is built on top of the run-time system. Chapter 7 covers high-level communication, while Appendix A contains the reference manual for the Strata communications package.

Strata was originally developed for Thinking Machines's CM-5 [TMC92] and also runs on the PROTEUS simulator [BDCW92].[1] Many of the original goals derived from inadequacies in Thinking Machine's low-level communication software, CMMD [TMC93]. Berkeley's CMAM [vECGS92][LC94] arose from similar considerations, but primarily focused on active messages. The current version of CMMD inherited much from CMAM; similarly, Thinking Machines is currently integrating many of the features of Strata into CMMD. Section 6.5 summarizes the differences between Strata, CMMD, and CMAM. Figure 6-1 shows the relative performance of Strata, CMMD and CMAM via a set of microbenchmarks; Strata is consistently the fastest, although its impact on application performance is even greater. The following sections cover the performance and features of Strata in detail.

6.1.1 Synchronous Global Communication

Strata provides a wide array of primitives for global communication, including barrier synchronization, combine operations (scans and reductions), and global broadcast. The CM-5 version uses a mixture of the control and data networks, while the PROTEUS version assumes only a data network. Strata generalizes CMMD's primitives in three key ways: first, Strata supports split-phase operations, which allow overlap of communication and computation and provide substantially more flexibility. Second, Strata provides additional operations useful for statistical modeling, and third, Strata supports better segmentation options for segmented scans.

Split-phase operations allow the processor to perform work while the global operation completes, which increases performance and flexibility. Strata provides split-phase versions of barriers and combine operations. The split-phase interface consists of a set of three procedures that respectively start the operation, test for completion, and block pending completion. For example, the split-phase barrier procedures are:

`void StartBarrier(void)`
: Notify other processors that this processor has reached the barrier.

`Bool QueryBarrier(void)`
: Test if all processors have reached the barrier.

`void CompleteBarrier(void)`
: Block until all processors reach the barrier, which is identical to spinning until `QueryBarrier` returns true.

A regular barrier can be implemented using `StartBarrier` followed immediately by `CompleteBarrier`. Other split-phase operations follow the same pattern and naming convention.

Although split-phase operations improve performance, their real advantage is flexibility. By allowing the user to control the blocking loop, Strata enables a variety of algorithms that are not

1. Robert Blumofe and I expect to port Strata to the Cray T3D later this year.

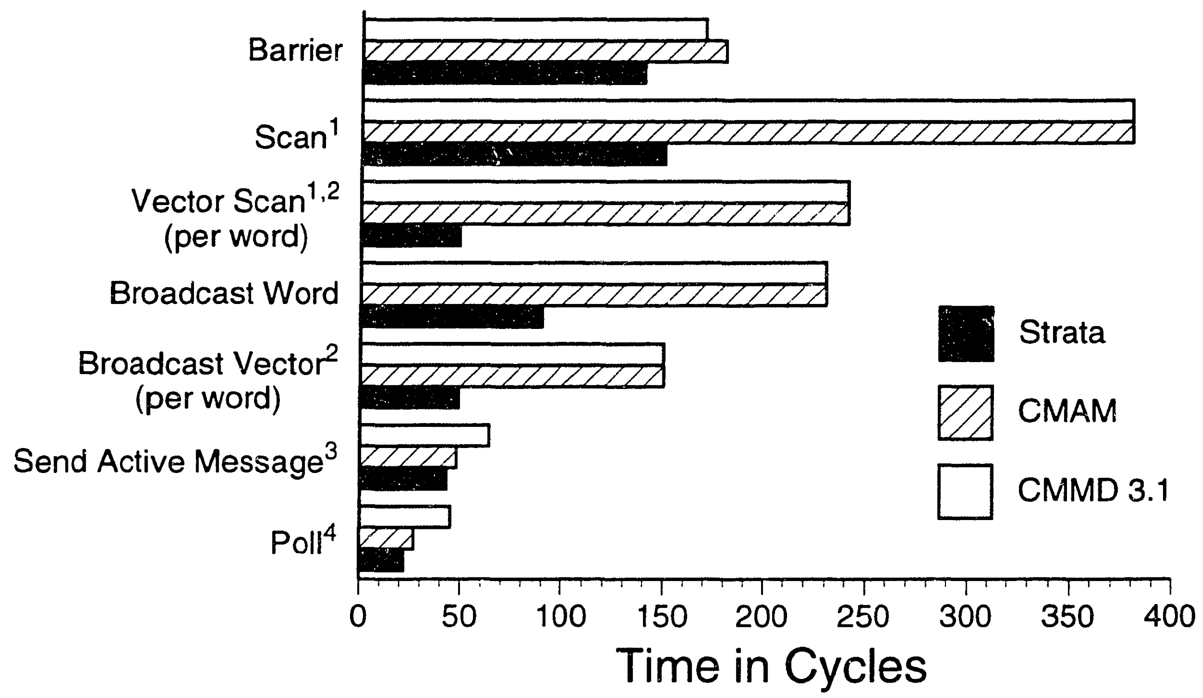


Figure 6-1: Relative Performance of CM-5 Communication Packages

The plot compares the performance of Strata, CMAM [LC94] and CMMD 3.1 according to a variety of microbenchmarks. Strata performs as well or better than the others in all cases, and is more than twice as fast for some operations. Notes: 1) scans are for unsigned integers, 2) vector length is 100 words, 3) Strata uses `SendLeftPollBoth`, CMMD uses `CMAML_request`, 4) Strata uses `PollBothTilEmpty`; all versions measure the cost with no arrivals.

possible under CMMD. For example, Chapter 7 shows how the use of barriers within bulk-communication operations improves performance substantially. The best performance requires both polling and a mixture of barriers and active messages. Without split-phase barriers this combination is not possible: regular barriers lead to deadlock. In the deadlock scenario, one processor completes sending messages and enters the barrier, which causes it to stop *accepting* messages until the barrier completes. At the same time, another processor stalls because the network is backed up due to messages going to the first processor. The second processor thus never enters the barrier and the application deadlocks. Split-phase barriers eliminate this problem by allowing the first processor to receive messages while it waits for the barrier to complete, which in turn allows the second processor to make progress and eventually reach the barrier. In general, split-phase operations allow the system to participate in other operations such as message passing while the "synchronous" global operation completes.

In addition to split-phase versions, Strata extends the set of reduction operations available to the user to support statistical modeling. Table 6-1 shows the CMMD and Strata reduction opera-

CMMD	Strata
Signed Add	Signed Add
Unsigned Add	Unsigned Add
Bit-wise OR	Bit-wise OR
Bit-wise XOR	Bit-wise XOR
Signed/Unsigned Max	Signed/Unsigned Max
Signed/Unsigned Min	Signed/Unsigned Min
	Average
	Variance
	Median

Table 6-1: Reduction Operations for CMMD and Strata

tions. The new operations are average, variance and median. These operations take one value from each processor and return the respective aggregate quantity, which simplifies data collection for statistical modeling. The implementation of the average operation is straightforward: a global add followed by a local division by the number of processors. The variance is similar and is based on:

$$\text{Variance} \equiv \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}\right) \tag{45}$$

with each summation corresponding to a global reduction. The median algorithm is more interesting and is given in Figure 6-2.

The third advantage of Strata's synchronous global communication primitives is extended support for segmentation. A segmented scan operation partitions the processors into contiguous groups by processor number; these groups are called segments. The combine operation occurs in parallel on each segment. Consider a segmented summation, with each segment denoted by a box:

$$\begin{array}{lcccc} \textbf{Input:} & \boxed{1\ \ 1\ \ 1} & \boxed{1\ \ 1\ \ 1\ \ 1\ \ 1} & \boxed{1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1} & \boxed{1} \\ \textbf{Output:} & 0\ \ 1\ \ 2 & 0\ \ 1\ \ 2\ \ 3\ \ 4 & 0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6 & 0 \end{array}$$

The output for a processor is the sum of the inputs of the processors to its left that are *in its segment*, which in this example equals the number of processors to the left (in the segment).

Implicit in this example is the requirement that each processor contribute exactly one value. An important generalization is the use of *virtual processors*. If we view the input as a vector, this generalization simply means that we are performing a combine operation on a vector that is wider than the number of real processors. The CM-5 hardware provides the more sophisticated segment boundaries required for virtual processors, but the CMMD software does not. Strata supports wide segments, but because the implementation follows from CM-5 hardware design, we consider this

```
Compute the median of $n$ values, one from each processor:

    Compute the average [global average]
    Find $p$, the smallest value greater than the average [global min]
    Count the number of values $\leq p$, [global sum]
    Mark count as "too big"
    While count $\neq \frac{n}{2}$ {
        If count was "too small" last iteration, but now is
            "too big", return $p$ [prevents infinite loop]
        If count $< \frac{n}{2}$,
            mark "too small"
            $p$ = next largest value [global min]
        else
            mark "too big"
            $p$ = next smallest value [global max]
        endif
        Count the number of values $\leq p$, [global sum]
    }
    Return $p$
```

Figure 6-2: Algorithm for Global Median

Pseudo-code for the global median operation. Global operations used as subroutines are indicated in square brackets. The basic idea is to start at the average and iterate toward the median, which is usually quite close.

advantage practical but not novel. Thus, we leave the details of segmentation for virtual processors to the Strata reference manual in Appendix A.

Both CMMD and Strata provide extensive support for global combine operations. However, Strata adds split-phase operations for performance and flexibility, additional operations that support statistical modeling, and sophisticated segmentation that exposes the full power of the hardware to the user.

6.1.2 Active Messages

Strata also provides the fastest and most complete set of active-message primitives. Active messages came out of dataflow research [ACM88] and were popularized by von Eicken *et al.* in their CMAM implementation [vECGS92] for the CM-5.

There are many options in the design of an active-message primitive. To begin with, the CM-5 actually contains two networks, the left network and the right network. The use of two logical networks supports deadlock-free protocols such as the request-reply protocol [Lei+92]. There are also three options for polling: do not poll, poll *n* times, and poll until empty. To completely specify a message-sending primitive, we must decide the network on which to send and the polling policy for both networks. Strata provides the complete set of options for sending and polling, nearly all of which are optimal in some situation.

Strata avoids interrupt-based message passing for two reasons. First, it is substantially slower due to the high overhead of interrupts on the Sparc processor. Second, it is much harder to build correct applications with interrupts, since interrupts require the system to use explicit locks for all critical sections, some of which may not be obvious. While interrupts imply that code is not atomic by default, the default behavior with polling is that handlers and straight-line code are atomic. Section 6.1.4 covers Strata's support for atomicity. The biggest argument in favor of interrupts is the freedom to avoid polling, which can simplify code and increase performance if most polls fail. However, this argument does not apply to the SPMD style in which all processors are in the same phase of the program. For example, processors need not poll during computation phases since it is clear that no processors will inject messages. For communication phases, the polls generally succeed and thus achieve much better performance than interrupts.

Strata uses protocols to avoid deadlock that achieve maximum performance. In particular, Strata is much more aggressive than CMMD or CMAM in using both networks, while still ensuring freedom from deadlock. The optimal protocols used by Strata were designed by Robert Blumofe and will appear in our upcoming paper on active messages [BBK94]. These protocols are optimal in the sense that they use both networks whenever possible.

Figure 6-3 plots the relative *application* performance of Strata against two variants of CMMD's active-message code. The application is a sparse-matrix solver that uses dataflow techniques to maximize parallelism; it is from a study coauthored with Frederic T. Chong, Shamik Sharma, and Joel Saltz [CSBS94]. The key conclusion is that Strata's active messages can provide up to a factor of two in application performance over CMMD. The benefits come from tighter code, optimal polling, and proper use of both sides of the network. Berkeley's CMAM should achieve performance close to that of `CMAML_request` since it only uses one side of the network. This application performs the vast majority of its work in handlers and thus should be viewed as an upper bound on the benefits of Strata's active-message layer.

Finally, it should be noted that originally the developers of CMAM argued against extensive use of handlers, preferring instead to create threads to perform the work of the application. The reasoning was essentially that it is too expensive to allow atomic handlers to perform real work, since they stop accepting messages and can back up the network. This application is the first that we know of that builds the entire application out of handlers. Given that the handler overhead is

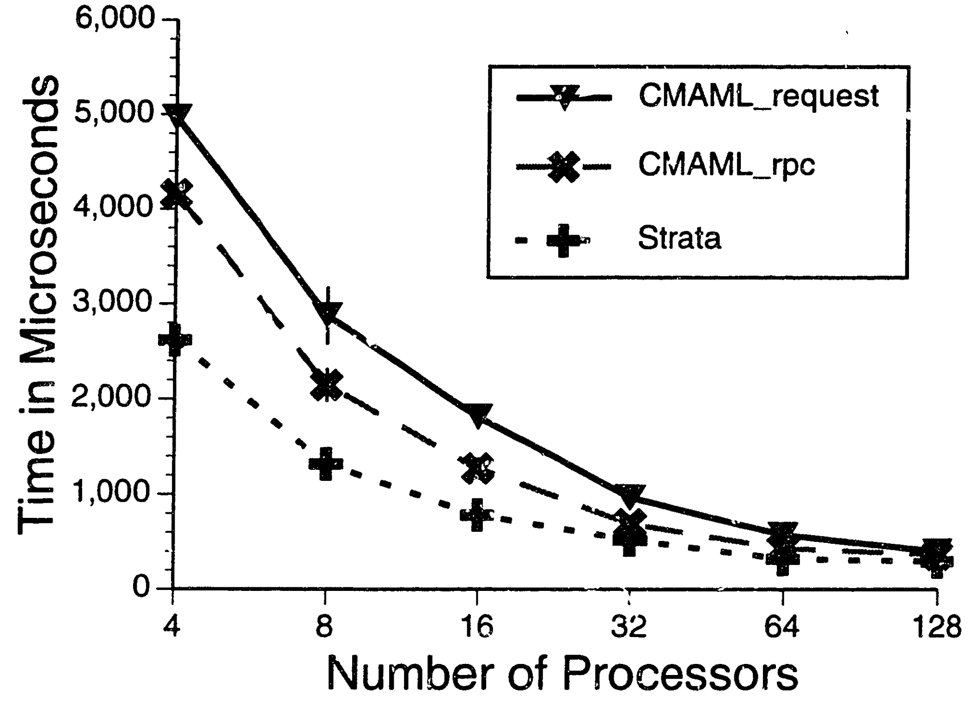


Figure 6-3: Relative Active-Message Performance

The graph shows the relative active-message performance for CMAML and Strata for a sparse-matrix solver running on a CM-5. CMAML_request performs the worst because it uses only one side of the network. CMAML_rpc uses both sides, but does not poll enough, which causes network congestion. Strata combines sufficient polling with both sides of the network to achieve the best performance. These curves reveal that Strata can improve *application* performance by a factor of two.

the limiting performance factor and that network congestion is not an issue, we can deduce that the thread-based version would perform significantly worse due to the additional overhead of thread management. Recent work on optimistic active messages [KWW+94] seeks the same benefit by moving work into handlers (optimistically), and then backing out in the case of deadlock or excessively long handlers.

6.1.3 Block Transfers and Bandwidth Matching

Strata's active messages stand out even more when they are combined to perform higher-level communication such as block transfers. Block transfers use *ports* to indicate the destination memory region. A port corresponds to a particular starting address and block size at the target node; this region is set up ahead of time by the receiver, so outside protocols must be used to ensure that the port is ready. Typically, all nodes will set up their ports and then enter a barrier; another common option is to use a round-trip message to set up the port.

The most common version of Strata's block-transfer primitives is:

```
void SendBlockXferPollBoth(int target, unsigned
                          port_num, int offset, Word
                          *buf, int size)
```

This routine sends the block pointed to by `buf`, which is `size` words long, to the indicated port on processor `target`. The `offset` argument is the offset within the port at which to store the incoming block; this allows multiple processors to fill in various parts of the port's memory. This particular flavor of block transfer sends packets on both networks and polls both networks.When all of a port's contents have arrived, an associated handler executes at the receiver. This gives block transfers the flavor of active messages.

Strata uses a novel flow-control mechanism to improve block-transfer performance. The key observation is that it takes about 62 cycles to receive a packet but only about 37 to inject a packet. Thus, the sender will quickly back up the network and then stall periodically to match the limited throughput of the receiver. Given that the sender must wait periodically, it is much better to wait explicitly and keep the network empty than it is to depend on stalling, which implies heavy network congestion. The resulting technique, called *bandwidth matching*, uses the following basic algorithm:

```
Loop until all packets sent {
    Inject a packet
    If there are arrived packets, handle all of them
    Otherwise, delay for 28 cycles
}
```

The optimal delay value, 28 cycles, is slightly more than the difference between the overheads for sending and receiving:

$$\text{Minimum Delay} = 62 - 37 = 25 \text{ cycles} \qquad (46)$$

Figure 6-4 shows the impact of bandwidth matching for 4-kilobyte blocks and compares Strata's block transfer to that of CMMD, which is called `CMAML_scopy`. The net gain with proper polling is about 25% more throughput.

There are several options for handling arrived packets. The option given in the pseudo-code above is to handle all of them, which we refer to as "poll until empty", since the receiver continues to handle packets until polling fails. The fastest option is to poll exactly once, regardless of how many packets are available; we refer to this strategy as "poll once". The advantage of polling only once is that it ensures that all nodes inject at nearly the same rate. In turn, this implies that on average there will be only one waiting packet, since each node alternates sending and receiving. The "poll until empty" version does almost as well, but wastes some time due to failed polls, which cost about ten cycles each.

Of particular interest in this context is the method used to determine the send and receive overheads: they are coefficients from the block transfer model and were produced by the auto-calibration toolkit! Thus, bandwidth matching is an example of how the models can provide insight into performance problems; in fact, the coefficients helped reveal the necessity of artificially limiting

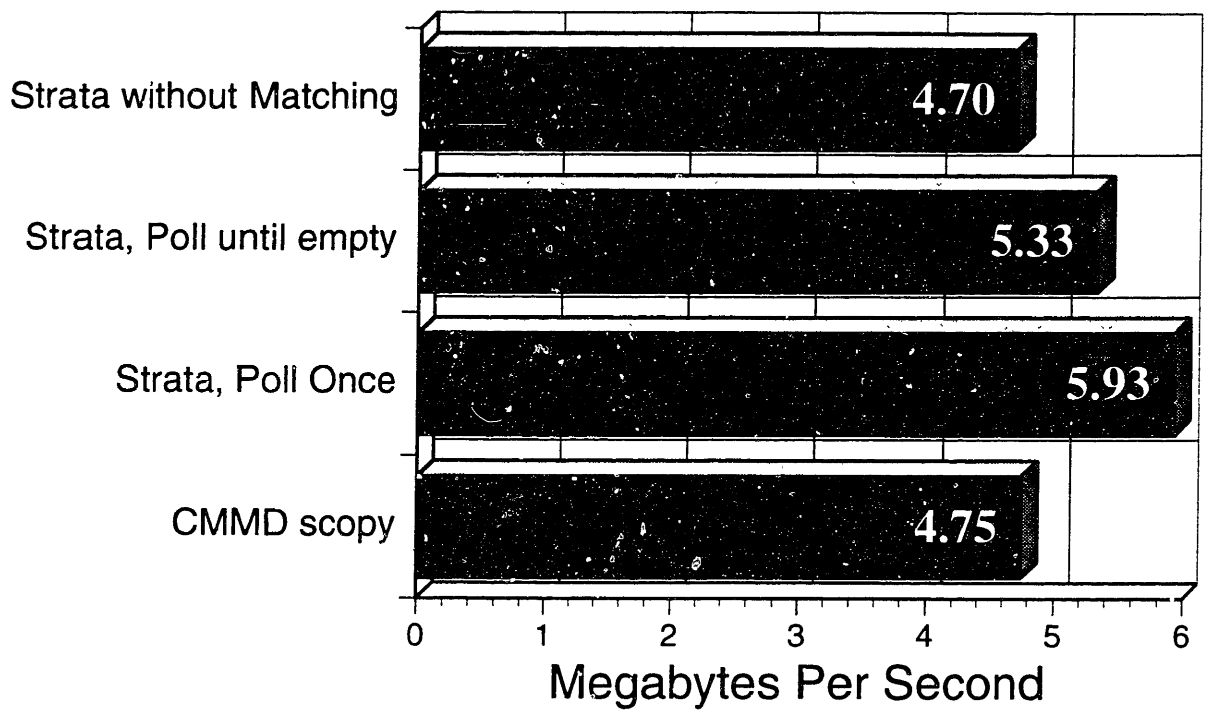


Figure 6-4: Relative Performance of Block Transfer
This plot shows the impact of bandwidth matching on block-transfer performance. The middle two bars use bandwidth matching, while the last bar shows CMMD for comparison. Bandwidth matching with optimal polling performs 25% better than CMMD's scopy routine.

the injection rate. Another key piece of evidence is the fact that Strata performs worse than CMMD without bandwidth matching, even though Strata's send and receive primitives are faster. In fact, CMMD performs slightly better *because* its primitives are slower! The slower primitives lead to fewer packets in the network and thus less congestion.

The data presented so far implies a limit of about 6 megabytes per second for the CM-5. In fact, this limit only applies to two-way transfers, those in which each processor is both sending and receiving. Given the combined overhead of about 89 cycles for the "poll once" case we get a predicted throughput of:[1]

$$\frac{16 \text{ bytes per packet}}{89 \text{ cycles}} \cdot \frac{33 \text{ cycles}}{1 \text{ microsecond}} = 5.93 \text{ megabytes per second} \qquad (47)$$

However, we can do much better for one-way transfers, which are limited only by the receive overhead, which is 62 cycles:

1. The combined overhead is less than $37 + 62 = 99$ because of there is only one poll instead of two.

$$\frac{16}{62} \cdot \frac{33}{1} = 8.52 \text{ megabytes per second} \tag{48}$$

To be clear, the bandwidth in equation (47) is 5.93 in *both* directions for a total of about 11.8 megabytes per second. The optimal bandwidth-matching delay for the one-way case is exactly 25 cycles, as given in equation (46). The delay for the two-way case is slightly higher to ensure that the receiver sees a window (of about three cycles) in which to inject a message; for the one-way case, we do not need such a window. Of course, the actual injection takes more than three cycles, but three is sufficient to ensure that the last poll fails and thus that the receiver begins sending. For the "poll once" case, there is no second poll and the extra delay has no effect.

For both polling versions, the normal case consists of alternating sends and receives. Bandwidth matching works exactly because it ensures that this pattern is stable and self-synchronizing; that is, the system naturally returns to this state. This explains the benefit of polling exactly once: we know that most of the time additional polls will fail and are thus pure overhead. However, "poll until empty" is more robust since it can recover from temporary congestion. For example, if a flaky network chip stalled packets the network could back up internally. The eventual burst of arrivals would be handled correctly by the "poll until empty" variant, but would lead to a stable queue in the "poll once" case, since the latter handles packets exactly as fast as they arrive. Although we have no evidence of such congestion, the released version of Strata defaults to "poll until empty" for block transfers on the basis of its robustness, even though we forfeit some performance.

Finally, note that the effectiveness of bandwidth matching strongly depends on the delay value, which is platform dependent. Traditionally, this would be a poor state of affairs, since it implies that new platforms are likely to inherit the suboptimal delay values of the previous platform. However, we can avoid this problem in an elegant manner. The models predict the delay value quite precisely; thus, the delay value can be set automatically based on the Strata models that are already generated during a port. This is just parameter optimization applied to the runtime system!

6.1.4 Debugging and Monitoring

Strata provides many facilities for debugging and monitoring, including the extensive support for timing discussed in Section 3.4. Three novel facilities are support for printing from handlers, support for atomic logging, and support for graphical output.

A common practical problem with active messages is the prohibition of printing statements within a handler. This restriction derives from the requirement that handlers execute atomically and from the non-atomic nature of output through the kernel. Strata provides replacements for C's `printf` and `fprintf` procedures [KR88] that atomically enqueue the output, which is then actually output via the kernel sometime after the handler completes. This mechanism, called `Qprintf`, solves the problem in practice quite nicely. Developers simply use `qprintf` whenever they want to add a debugging statement to a handler. The output occurs whenever most non-atomic Strata routines execute or it can be flushed explicitly by the user via `EmptyPrintQ`. There is also support for viewing queued-up output from within a debugger such as `dbx` [Lin90], which handles the case when a handler has been executed, but the resulting output is still pending. Despite its simplicity, this is probably the most popular mechanism in Strata.

Strata also provides extensive support for logging and tracing, which can be turned on during debugging. First, all of the global operations can log their state. The resulting log can be used to determine the cause of deadlocks involving global operations, which are a common form of error in the SPMD model. For example, if one processor forgot to enter the barrier, the log reveals that the last entry for the rest of the processors is essentially "waiting for barrier", while the missing processor has some other entry. Another common variant occurs when two global operations take place simultaneously, in which case the log reveals the operations and which processors participated in each one. Users can also log application-specific information using a variant of `printf` called `StrataLog`.

The logging is atomic so that the log always reflects the current state of the machine. Logging is implemented by sending data over the diagnostic network of the CM-5. This network is the right choice despite its very low bandwidth, because the operating system ensures that the output is atomically saved to a file. Without logging, these errors are usually found by polling processors by hand to see where they are, which is time consuming and prone to errors.

The final novel debugging facility of Strata is support for graphical output. Although Thinking Machines provides several debugging tools, none of them provide a global view of the application. The Strata approach matches that taken for the PROTEUS simulator: generate a trace file during execution and use an outside tool to convert it into a graphical representation of the application's behavior. In fact, Strata and PROTEUS use the same trace-file format and graph-generation program [BD92].

In the normal case, each processor tracks its progress among several user-defined states: the trace file contains timestamped state transitions. The graphing tool, called `stats`, converts these into a *state graph*, which is a three-dimensional graph of states versus processor numbers versus time. Figure 6-5 shows a state graph for the radix-sort program from Chapter 4. The graph reveals the relative length of each of the phases; in particular, the dominant cost is due to reordering the data. The graph also shows that there are seven iterations, which implies that there must be seven digits (which there were). Finally, we note that processor zero took substantially longer during initialization, which results from its management of the output file.

Strata facilities for debugging greatly enhance the development environment of the CM-5 for low-level application development. Printing from handlers, atomic logging, and state graphs have proved invaluable for the kinds of problems that arise in parallel programming. The next section looks at Strata on PROTEUS, which has even better support for development and thus was the development platform for most of the high-level library applications.

6.2 Strata for PROTEUS

The Strata communication package also runs on the PROTEUS simulator [BDCW92]. For simplicity and diversity the PROTEUS version is significantly different in implementation than the CM-5 version. The biggest difference is the absence of a control network; PROTEUS uses the data net-

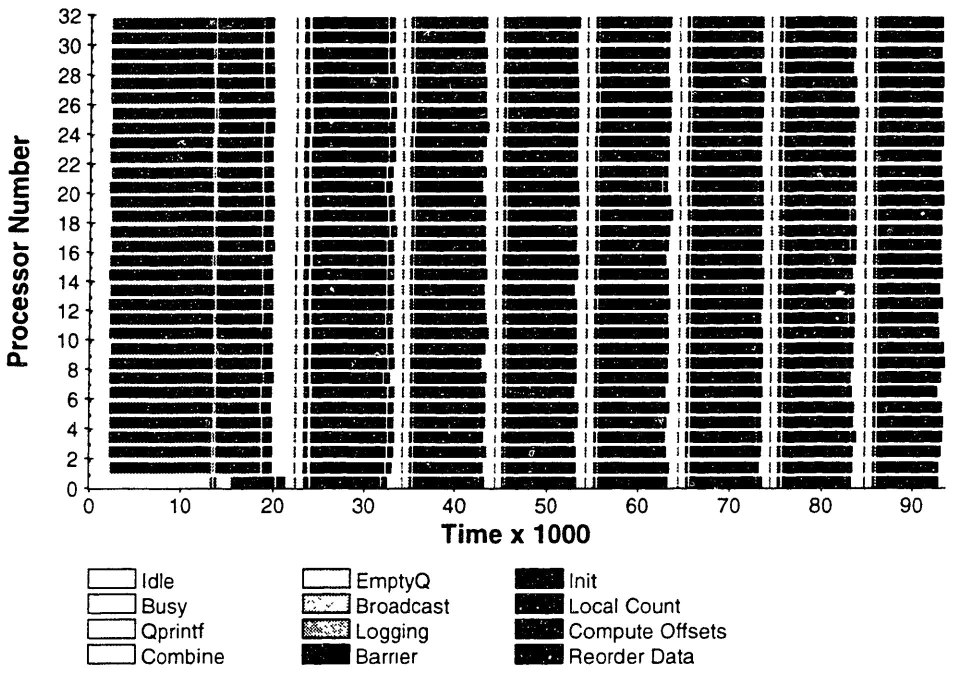


Figure 6-5: State Graph for Radix Sort

This graphs plots the state of each processor versus time, which is particularly useful for SPMD programs. The initialization phase and seven iterations of the main loop are visible. The majority of black reveals that this version spends most of its time reordering the data.

work for all of the global synchronization operations. The PROTEUS versions use a MIPS processor rather than a Sparc and have a completely different operating system. These differences combine to form a challenging test of both Strata and the auto-calibration toolkit; it is somewhat surprising that the models of Figure 6-6 had so little trouble tracking both versions from the same model specification.

The global synchronization operations in Strata use an explicit combining tree based on 20-word active messages. For simplicity, the vector combine operations are not pipelined, which leads to the appearance of `length*logP` terms in the `CombineVector` model. The use of 20-word active messages is itself an important difference: the CM-5 at MIT has a limit of four words per active message.

PROTEUS provides a development environment greatly superior to that of the CM-5 or any other large multiprocessor. The advantages of development via simulation are covered in detail in a paper coauthored with William Weihl that appeared in the 1993 Workshop on Parallel and Distributed Debugging [BW93]. The cross-development of Strata applications on PROTEUS represents the most complete implementation of this methodology and exploits most of the advantages.

First, PROTEUS supports a perfect view of global time. User functions can be timed exactly with zero overhead, so there are never measurement errors. Second, PROTEUS is deterministic by default, which means that it provides repeatability for easy debugging. Repeatability is priceless: bugs that appear occasionally or only in the absence of debugging code are nearly impossible to track down. PROTEUS also provides nonintrusive monitoring and debugging, which when combined with repeatability, ensures that bugs reoccur even in the presence of new debugging code. PROTEUS also benefits from the superior tools available on modern workstations; multiprocessors like the CM-5 tend to suffer from a dearth of quality tools due to their inherent youth and small numbers.

Thus, the methodology for the development of both Strata and the applications it supports is to first get the application running on PROTEUS, and then to port it to the CM-5. The CM-5 version almost always runs correctly the first time, since the run-time system is identical. On the few occasions when the CM-5 versions revealed bugs that were not obvious on PROTEUS, the primary cause was changes in timing that exercised a new part of a nondeterministic program. Removing these bugs on the CM-5 was simplified by the knowledge that the problem was timing dependent and was therefore probably a simple race condition. Although impossible to quantify, the impact of PROTEUS development environment was very substantial: there were many bugs that would have been far more difficult to resolve without PROTEUS' tools and repeatability.

6.3 Modeling Strata

An important property of Strata is its complete set of accurate performance models, more than thirty in all. Most of these are generated automatically by the auto-calibration toolkit. Because of its low-level nature, a few of the Strata routines require their own data collection code, but the toolkit handles the rest of model generation. For example, it is tricky to measure the cost of receiving an active message, since placing timing code in the handler ignores much of the cost. For this case, the calibration code times a loop with and without 100 incoming packets: the difference in times is the cost of receiving 100 packets. The time is converted into a cost per packet and then output just like any other sample; later stages of the toolkit cannot distinguish which samples were generated by custom data-collection code.

Figure 6-6 shows some of the more interesting Strata models for four different platforms, the CM-5 and three PROTEUS versions with varying communication costs. The `ClearMemory` costs are the same for each of the PROTEUS versions, since only the communication overhead varies among them. For `CombineVector`, the ATM stands out because of its large start-up cost, which is denoted by the first term. The CM-5 also stands out because the model is independent of the height of the combine tree, which is denoted by the absence of a `logP` term. This occurs because the CM-5 pipelines vector scans, so that the height of the tree affects only the depth of the pipeline, which has negligible impact on the overall cost. The round-trip cost varies about as expected except for the appearance of a small `logP` term in the Alewife model, which is probably capturing some congestion due to the limited bisection bandwidth for larger machines. The interrupt delay is high for the CM-5 because of the Sparc's register windows; the three PROTEUS versions are based on a MIPS R3000 processor, which does not have register windows.

Implicit in the existence of these models is the fact that they are robust to architectural and operating-system changes: the same list of model terms was used for each of the four platforms. The low MRE values, which follow the colon after each model, confirm that the toolkit success-

```
ClearMemory(length):2.33 =
    69.28 + 1.005*length

CombineVector(length):1.71 =
    110.1 + 47.5*length

Roundtrip(hops):0.41 =
    404.2 + 14.4*hops

SendRequest():0.02 = 68

SendRPC():0.03 = 44

InterruptDelay():0.06 = 70
```

CM-5 Models

```
ClearMemory(length):1.46 =
    66.59 + 1.382*length

CombineVector(length):0.48 =
    181.5 +75.98*length
     + 288.3*logP
     + 89.05*length*logP

Roundtrip(hops):0.51 =
    207.6 + 2.552*hops

SendRequest():0.08 = 33

SendRPC():0.11 = 33

InterruptDelay():0.07 = 5.33
```

Alewife Models

```
ClearMemory(length):1.51 =
    66.58 + 1.383*length

CombineVector(length):0.69 =
    203.9 +78.23*length
     +332.7*logP
     +89.82*length*logP

Roundtrip(hops):0.98 =
    302.5 + 11.23*hops

SendRequest():0.147 = 143

SendRPC():0.062 = 142.3

InterruptDelay():0.07 = 5.33
```

Paragon Models

```
ClearMemory(length):1.41 =
    66.59 + 1.383*length

CombineVector(length):1.56 =
    2678 + 68.51*length
     + 122.7*length*logP

Roundtrip(hops):0.96 =
    2032 + 61.89*hops

SendRequest():0.21 = 1130

SendRPC():1.04 = 1131

InterruptDelay():0.07 = 5.33
```

ATM Models

Figure 6-6: Four Versions of a Subset of the Strata Models

These four sets of models cover a wide range of underlying hardware costs. The Alewife, Paragon and ATM versions are simulated by PROTEUS. The number following the colon for each model gives the MRE.

fully generated the models despite the wide variance of the underlying costs. Finally, note that there is a symbiotic relationship between Strata and the toolkit: Strata provides the timing and data collection facilities required by the toolkit, while the toolkit produces models of Strata that lead to its improvement and to better use of Strata by higher-level abstractions and applications.

6.4 Conclusions

Strata offers many advantages over other low-level communication layers, and represents the leading edge in many areas. Novel features include:

- ❑ Strata provides a fast, safe split-phase interface for synchronous global communication, with many operations and very general segmentation options. The split-phase versions allow complicated protocols without deadlock, which includes mixing global operations with message passing. This will be critical for the next chapter, where we use barriers to improve the performance of bulk communication operations.
- ❑ Strata's active messages are extremely fast: we obtained speedups of 1.5 to 2 times for a realistic sparse-matrix application. Strata's active-message performance comes from efficient implementation, effective use of both sides of the network, and flexible control over polling.
- ❑ The block-transfer functions achieve the optimal transfer rate on the CM-5, reaching the upper bound set by the active-message overhead. The key to sustaining this bandwidth is a novel technique called bandwidth matching that acts as a static form of flow control with essentially zero overhead. Bandwidth matching increases performance by about 25%; as we will see in the next chapter, it also greatly reduces the variance in execution time of complex communication patterns.
- ❑ Strata's support for development includes printing from handlers, atomic logging, and integrated support for state graphs that provide insight into the global behavior of a program. By using PROTEUS as the primary development platform, Strata users can exploit the mature tools of their workstation as well as the repeatability and nonintrusive monitoring and debugging provided by PROTEUS. This environment greatly reduced the development time for the high-level library applications.
- ❑ All of the Strata primitives have accurate performance models that were generated in whole or in part by the auto-calibration toolkit. These models lead to better use of Strata as well as improvements in the Strata implementations. Strata also provides timing and statistical operations that support the data collection required for automatic model generation.

6.5 Related Work

The technique known as "active messages" has a somewhat complicated history. The first version of short messages with actions at the receiver was developed by dataflow researchers, led by Arvind [ACM88]. Bill Dally incorporated the technique into the Message-Driven Processor, which is the processor in the J-Machine [Dal+89]. Dataflow architectures like Monsoon [TPB+91] also provided hardware support for handler invocation. PROTEUS [BDCW92] based its communication on handlers because it was the most general mechanism and because it was required to simulate the J-Machine. Concurrently, David Culler brought the dataflow influence to Berkeley and started the TAM project ("threaded abstract machine"), which led to the active-messages paper [vECGS92] and brought the technique to commercial supercomputers including the CM-5. Berkeley's active-message layer for the CM-5, called CMAM, was integrated into CMMD 3.0 by Thinking Machines to produce TMC's CMAML [TMC93].

Strata's active-message layer was influenced by both CMAML and PROTEUS' message-passing code. The original motivation for providing a new active-message library was Robert Blumofe's work on better deadlock-free protocols [BBK94], and the importance of using both sides of the network.

Guy Blelloch investigated the scan model of computation and influenced the scan hardware of the CM-5, in part through his advisor Charles Leiserson. Blelloch's thesis investigates scans as the basis of data-parallel computing [Ble90]; a journal version appeared in *IEEE Transactions on Computers* [Ble89]. The follow-on work at Carnegie Mellon University is the Scandal Project, which includes the NESL language [BCH+94], a Common Lisp based language for nested data parallelism that uses scans as a key primitive. The implementation uses a portable scan and vector library written in C, called CVL [BCH+93]. When the CM-5 version of CVL was changed to use Strata, NESL's performance increased by about 40%.

The importance of accurate timing became apparent through the use of PROTEUS, which provides perfectly accurate timers for free as a by-product of simulation. The CM-5 provides 32-bit cycle counters in hardware [TMC92]. In addition, they are globally correct, which was probably unintentional, although it resulted from good design methodology. In particular, the CM-5 uses a globally synchronized clock with controlled skew to simplify inter-chip signalling, and the clock and reset lines follow the fat-tree structure to simplify system scaling. In turn, this means that the reset signal reaches all leaves (nodes) with exactly the same clock skew, which implies that the all of the cycle counters are reset on the same cycle, regardless of the size of the machine. Strata exploits this undocumented property to provide its globally synchronous measure of time.

The MIT Alewife machine [Aga+91] provides a 64-bit atomic-access cycle counter with the bottom 32 bits in hardware. Providing a 64-bit low-latency atomic cycle counter is not trivial; Alewife's was designed primarily by John Kubiatowicz with some help from me. Alewife also uses a globally synchronous clock, and we were careful to ensure that Alewife provides globally consistent 64-bit timers, exactly suitable for Strata.

The Alpha microprocessor from Digital Equipment Corporation also provides a 32-bit cycle counter in hardware [Sit92]. With operating-system support, it can emulate a 64-bit atomic cycle counter. It has the advantage that it can also time only the cycles within a particular thread, which avoids Strata's problem of counting *everything*.

There are several current parallel systems that provide some form of sequential debugging environment, as promoted by development via simulation for Strata [BW93]. The LAPACK library for linear algebra provides a sequential version for workstations [And+92]. High-Performance Fortran [HPF] and pSather [FLR92] are parallel languages that have sequential versions intended for development. The Cray T3D comes with a sequential emulator for development [Cray94]. There are also some debugging tools that can replay a particular parallel execution by logging race-condition results as the program executes [LM86][NM92].

7

High-Level Communication

This chapter looks at communication patterns as a whole in order to develop techniques that achieve high performance and predictability. We develop several novel techniques and show how they improve performance over the naive use of block transfers. We also develop a set of modules on top of Strata that implement complex patterns with near-optimal performance and improved predictability.

The key to effective high-level communication is the elimination of congestion; all of the techniques eliminate congestion in some way. In practice, the primary cause of congestion is insufficient bandwidth at some particular receiver. This occurs either because the receiver is not accepting messages or because packets arrive faster than the bandwidth limitation imposed by the receiver's software overhead. As we saw in the last chapter, bandwidth matching is effective because it eliminates this source of congestion for one-on-one transfers. Although bandwidth matching handles the case of a single sender-receiver pair, it provides no help for many-to-one communication patterns. This chapter is primarily about techniques to implement complex communication patterns as a sequence of concurrent one-to-one transfers on top of Strata's bandwidth-matched block transfers. Most of the contents of this chapter appeared in a joint paper with Bradley Kuszmaul at the 1994 International Parallel Processing Symposium [BK94].

7.1 The CM-5 Network

The new techniques have been implemented on two different networks, the CM-5's fat tree and the direct networks simulated by PROTEUS. However, because it is difficult to simulate congestion accurately, this chapter concentrates on the CM-5 for detailed performance analysis of the various techniques.

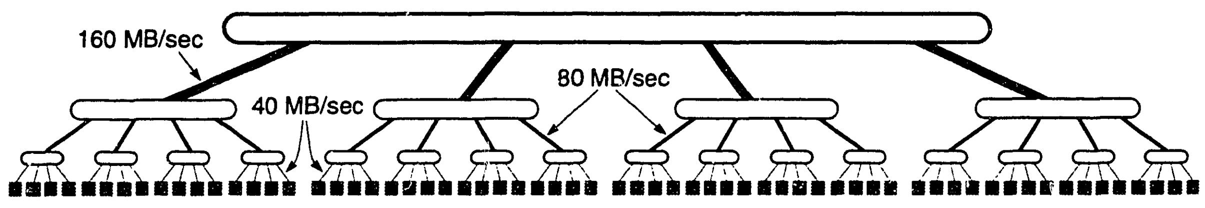


Figure 7-1: The CM-5 Fat Tree

The CM-5 network is a 4-ary fat-tree with 40 megabyte per second (MB/sec) links to the nodes, 80 MB/sec links to the second level switches, and 160 MB/sec links to the third level switches. The links represent the total bidirectional bandwidth. For a 64-node CM-5, this provides a unidirectional bisection bandwidth of 160 megabytes per second. This figure is based on a drawing by Bradley Kuszmaul.

The CM-5 uses a fat-tree network [Lei85], which is an indirect network shaped like a tree that uses higher-bandwidth ("fatter") links for the upper levels. The high-bandwidth links ensure that the bandwidth from level to level remains about the same even though the number of logical edges decreases as you move up the tree. Figure 7-1 shows the topology of the CM-5's fat-tree network for a 64-processor machine.

For the 64-node CM-5, the first level, which connects processors to switches, has an aggregate bandwidth of $64 \times 40 = 2560$ megabytes per second. The second level links are twice as fat, but there are one-fourth as many, for a total of 1280 megabytes per second. The top level links are twice as fat again, but there are one-sixteenth as many links as in the first level, which gives an aggregate bandwidth of 640 megabytes per second. The bisection bandwidth, which is the bandwidth across the minimum cut that partitions the processors equally, is 320 megabytes per second, or 160 megabytes per second in one direction. (In the figure, the minimum cut goes right down the middle.)

The bottleneck in practice is almost always the software overhead, which limits the bandwidth to 11.86 megabytes per second per node, or 5.93 in one direction, as shown in equation (47) in Section 6.1.3.[1] Multiplying by 64 processors, we find that the maximum achievable bandwidth is about 380 megabytes per second, which slightly exceeds the bisection bandwidth. Thus, for the CM-5 the bisection bandwidth is rarely an issue, even for worst-case communication patterns.

The CM-5 uses adaptive routing on the way up the tree, selecting pseudo-randomly from the available links. Although logically there is only one correct direction for each hop, the switches

1. This bandwidth limit is based on the best-case instruction sequence from Strata, and can also be measured directly. It assumes that the data must be stored at the receiver, and that enough data is sent to make the fixed start-up overhead negligible (a few kilobytes). Higher bandwidths would be achieved if the receiver could avoid writing to memory, such as when computing a running sum. However, it is difficult to construct a scenario that requires that all of the data be sent (rather than sending the summary statistic directly), without requiring the receiver to store it even temporarily.

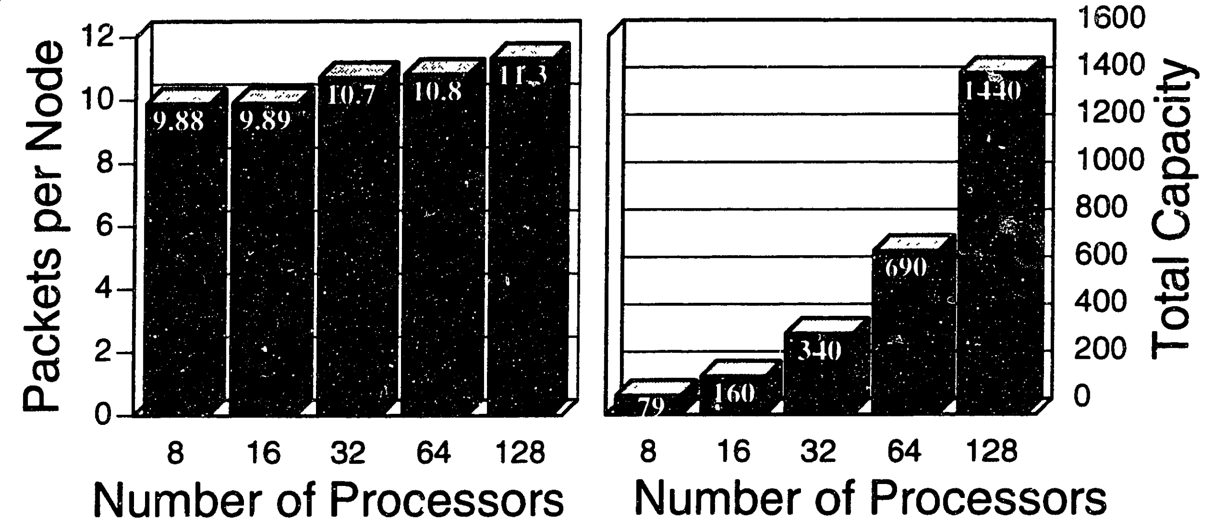


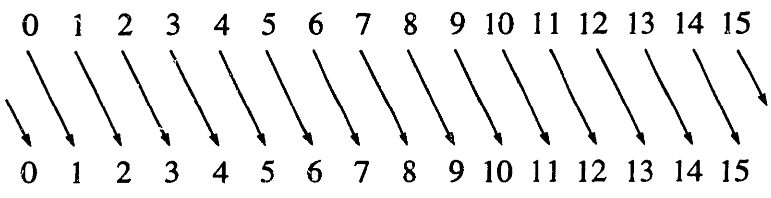
Figure 7-2: CM-5 Network Capacity

These plots show the capacity of the CM-5 network. The left plot shows how many packets each node can inject on average when none of the processors are accepting packets. The count increases slightly at 32 and 128 because those values require an additional level in the fat tree over the next smaller machine size. The plot on the right gives the total capacity of the network.

actually consist of many equivalent chips, so that the chip within the switch is selected randomly. Messages rise only as high as required to reach their destination, which is to the lowest common ancestor. On the way down there is one logical path and also one physical path, so routing is deterministic. By using randomness on the way up, the network ensures even load across the root switches; in turn, this even load ensures even distribution of packets on the way down, assuming uniform distribution of the target processors. Thus, in the case of uniform distribution, the CM-5 network essentially eliminates congestion. Of course, the network can not significantly reduce congestion induced by many-to-one communication patterns, since the bottleneck is the receiver rather than the network.

Another important aspect of the network is its *capacity*, which is the number of packets that can be injected without the receiver removing any. The capacity limits the ability of processors to work independently, since the sender stalls if the network is full. For example, if the network can hold ten packets, then a node can only inject ten packets before it must wait for the receiver to remove some. Figure 7-2 plots the capacity of the CM-5 network for a variety of machines sizes. Each node can inject about ten packets before it must wait for the receiver. The network essentially acts as a buffer with a capacity of about ten packets per block transfer, which is about 30 microseconds for the two-way transfers used in this chapter.[1] Thus, for globally choreographed patterns, the processors must stay within 30 microseconds of one another to avoid congestion.

1. The buffering in terms of time is calculated from the bandwidth-matched bidirectional transfer rate of about 3 microseconds per packet times 10 packets.



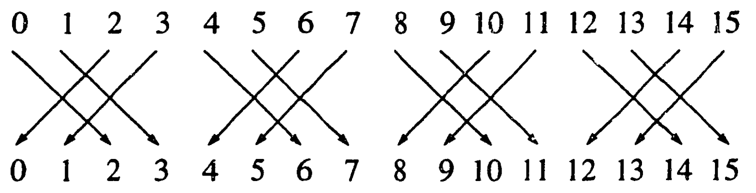


Figure 7-3: Example Permutations for 16 Processors

The top permutation, commonly called CSHIFT, uses modular addition to compute the target address. In the bottom permutation, the operation is XOR with the constant 2, which flips the second least significant bit.

Having completed a review of the CM-5 network, we now look at the software techniques required to maximize the effective bandwidth.

7.2 Permutation Patterns

Given that we wish to avoid many-to-one communication patterns, the easiest place to start is with permutations, which by definition are one-to-one. Permutations are often used in supercomputer applications, and are particularly important here, since they form the basis of our modules for complex communication patterns.

One of the most important permutations is the cyclic shift, or “CSHIFT” (pronounced “SEE-shift”), which is shown in Figure 7-3. The cyclic shift permutation computes the target address using modular addition, with the number of processors as the modulus. For example, a cyclic shift by four on a 64-node machine computes the target node number via:

$$\text{target}(i) = (i+4) \bmod 64 . \qquad (49)$$

Cyclic shifts are commonly used for nearest neighbor communication; the stencil applications of the previous chapters used cyclic shifts.

Another common permutation is the hypercube exchange, in which nodes exchange information along one dimension of a virtual hypercube. In the figure the exchange occurs along dimension one, which corresponds to a permutation of:

$$\text{target}(i) = i \text{ XOR } 2. \tag{50}$$

Although originally used to exploit the links of a hypercube network, this permutation is still common today in scientific applications. Both of the permutations in Figure 7-3 could be used as the basis of complex patterns, such as all-pairs communication.

The use of permutations is critical to high-performance communication. We can estimate the impact of using permutations for SPMD code, in which the time of a phase is the time of the slowest node. If we assume that each processor has one block to send (of a fixed size), that there are n processors, and that targets are uniformly chosen at random, then we can compute the expected maximum number of packets sent to one processor. This is just a version of the classic n balls in n bins problem, which has the well known result [GKP89]:

$$\text{Expected Maximum} = \Theta\left(\frac{\log n}{\log\log n}\right). \tag{51}$$

Although difficult to compute for a particular n, we can measure the maximum for a series of random trials. Figure 7-4 shows the number of times each maximum value occurred in 1000 trials for $n \equiv 64$. The expected value of the maximum is 4.3 transfers, compared with just one for a permutation. Since these transfers are completely sequential, we expect at least one processor, and thus the entire phase, to take about four times longer using random targets instead of a permutation. Note that this effect is really just congestion at the receiver due to packet collisions; we refer to multiple transfers to the same node as *target collisions* to distinguish them from internal network collisions.

Figure 7-5 plots the relative performance of the three target distributions for four-kilobyte block transfers. In this experiment, each node sends a four-kilobyte block transfer to one node, which is selected according to the distribution. The "Random Targets" distribution picks nodes uniformly at random. The other two are permutations: the "Exchange" distribution computes targets according to equation (50), while the "Cyclic Shift" distribution is a cyclic shift by four. As expected, the random distribution does almost four times worse. Given that we can effectively implement permutations, we next look at how to combine them into more complicated patterns.

7.3 Sequences of Permutations: Transpose

The key idea in this section is quite simple: to get good performance for complex patterns, we decompose them into a sequence of permutations separated by barriers. Although the barriers clearly add overhead, they lead to significantly better performance by eliminating congestion. In addition, barriers lead to very predictable performance, since we can just sum up the times for the individual phases and add in the fixed overhead for the barriers themselves.

The key result is that the addition of barriers to a sequence of permutations improves performance on the CM-5 by a factor of two to three. This use of barriers was first noticed by Steve

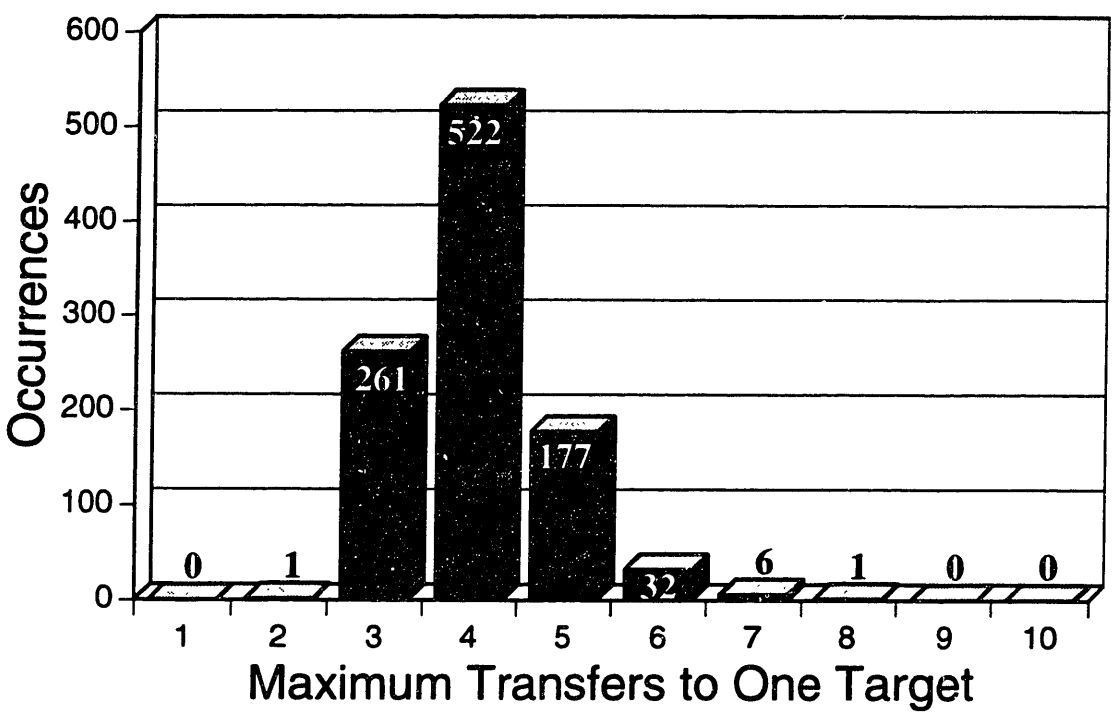


Figure 7-4: Histogram of Maximum Transfers to One Processor
This graph shows the number of times in 1000 trials that each value was the maximum number of transfers to one processor in a 64-node machine. The expected maximum value is 4.3, while more than 99% of the time at least one processor has at least three incoming messages.

Heller of Thinking Machines, and Culler *et al.* mention the effect in a later paper [CKP+93]; neither group documented the effect or established the causes behind it.

The benchmark is the transpose pattern [Ede91], in which all pairs must communicate some fixed size block in both directions. We implement the pattern as a sequence of cyclic-shift permutations, using each of the 64 possible offsets exactly once. Figure 7-6 shows the relative performance of transpose implementations for different block sizes, but with the same total amount of communication, so that we send eight times as many 128-byte blocks as we do 1K blocks. The two versions without barriers, one for Strata and one for CMMD, perform the worst by far. The CMMD curve represents the effective bandwidth actually achieved by the vast majority of users!

Except for very small block sizes, the implementations with barriers perform significantly better. At 64-byte blocks, which is only four packets, the difference is small, but by 128-byte blocks the difference is a factor of two. For even bigger blocks, adding barriers to the CMMD version brings the bandwidth from about 1.7 megabytes per second to around 3.6. However, this performance is not possible with CMMD alone: it requires Strata's barriers. The barriers must be split-phased to ensure that finished nodes continue to accept messages; using the barriers provided with CMMD leads to deadlock.[1] However, even with Strata's barriers, the performance lags the Strata-

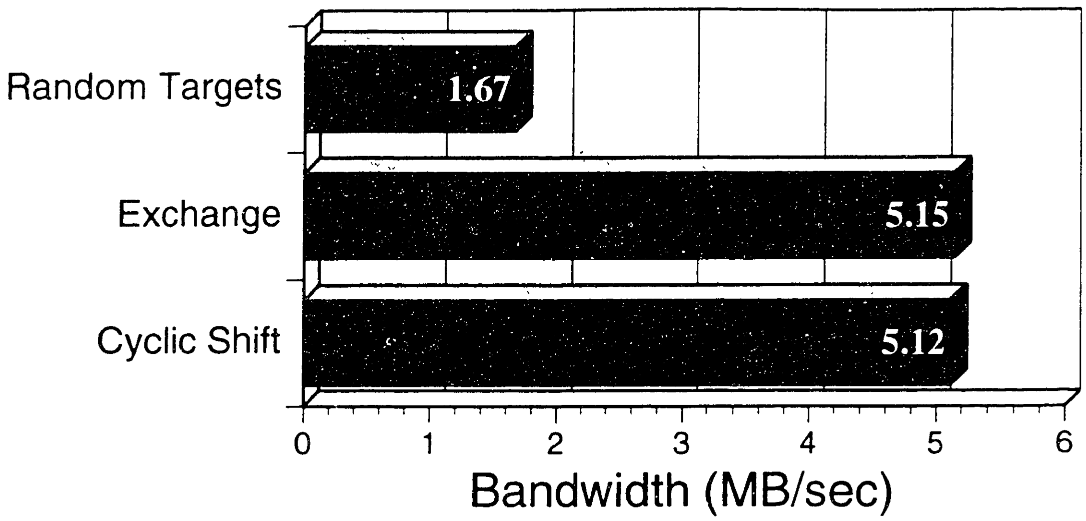


Figure 7-5: Bandwidth for Different Target Distributions
This graphs shows the effective bandwidth for a 64-node machine in which each node picks a target for a 4K block transfer according to the specified distribution. The permutations approach the bandwidth limit of 5.3 megabytes per second, while the random distribution takes about 3.6 times longer.

only versions considerably. Furthermore, the 95% confidence intervals for the "CMMD with (Strata) Barriers" version are huge: the performance is very difficult to predict.

7.3.1 Why Barriers Help

Before we investigate this figure further, it is important to examine the cause behind the large impact of barriers. Since we know that a single permutation behaves well, the bandwidth degradation must be due to some interaction among the permutations. Since packet reception is known to be the bottleneck for block transfers, even a single target collision can quickly back up the network. Of course, once the network is congested some senders will stall, which affects the completion time for a permutation. In turn, this leads to multiple permutations in progress simultaneously, which causes more target collisions, since a node is the target for two different sources in different permutations.

To observe target collisions, we measured, at each instant in time, the number of packets in the network that are destined for a given processor. We did this by recording the time that each packet was injected into the network and the time that the target received the packet. We used Strata's globally synchronous cycle counters to obtain consistent timestamps. Figure 7-7 shows evidence for target collisions for one typical case: cyclic shifts with no barriers and a block size of

1. The CMMD barriers could be used with interrupt-based message passing. However, since the interrupt overhead is at least as much as the active-message overhead, the end performance would be no better and probably would be worse.

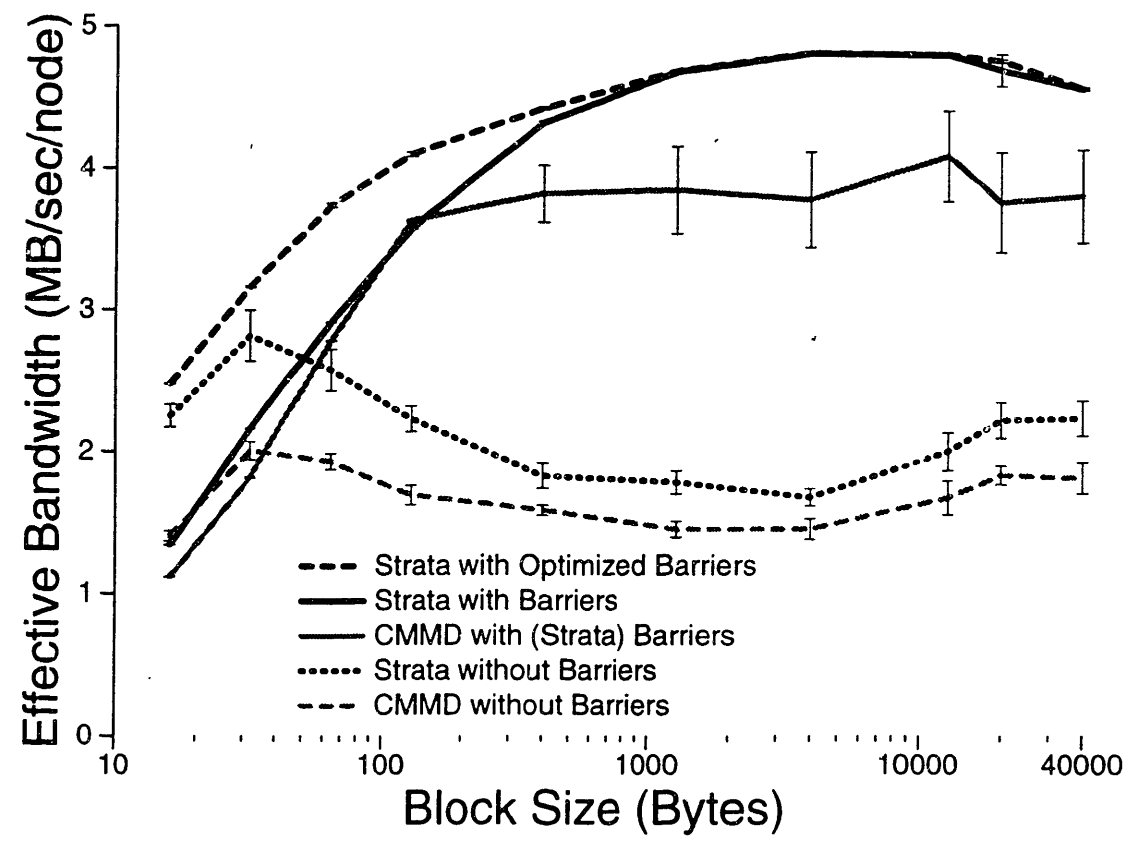


Figure 7-6: Combining Permutations with Barriers

This graph shows the effect of barriers on sequences of permutations, in this case cyclic shifts. The lines mark the average bandwidth; the error bars denote the 95% confidence intervals. The barrier versions are 2-3 times faster. The low variance for the upper Strata curves is due to bandwidth matching.

100 packets, or 1600 bytes of data. The offset starts at zero and increments for each permutation. The graph shows for each processor, at each point in time, the number of packets destined for that processor.

There are several interesting patterns in this data. At the far left there are some patterns that appear as a "warped checkerboard" around 200,000 cycles. These patterns reflect the propagation of delays: a single packet was delayed, which caused the destination processor, which is two above at this point, to receive packets from two different senders. This in turn delays the receiver's injection of packets and thus exacerbates the problem. In short order, processors alternate between being overloaded (gray) and idle (white). By 400,000 cycles, some processors have as many as 17 packets queued up. The processors above the heavily loaded processors are nearly always idle and thus appear white. These processors are idle because several senders are blocked sending to their predecessor in the shift, which is the heavily loaded processor. The white regions thus change to black in about 100,000 cycles as the group of senders transition together. These

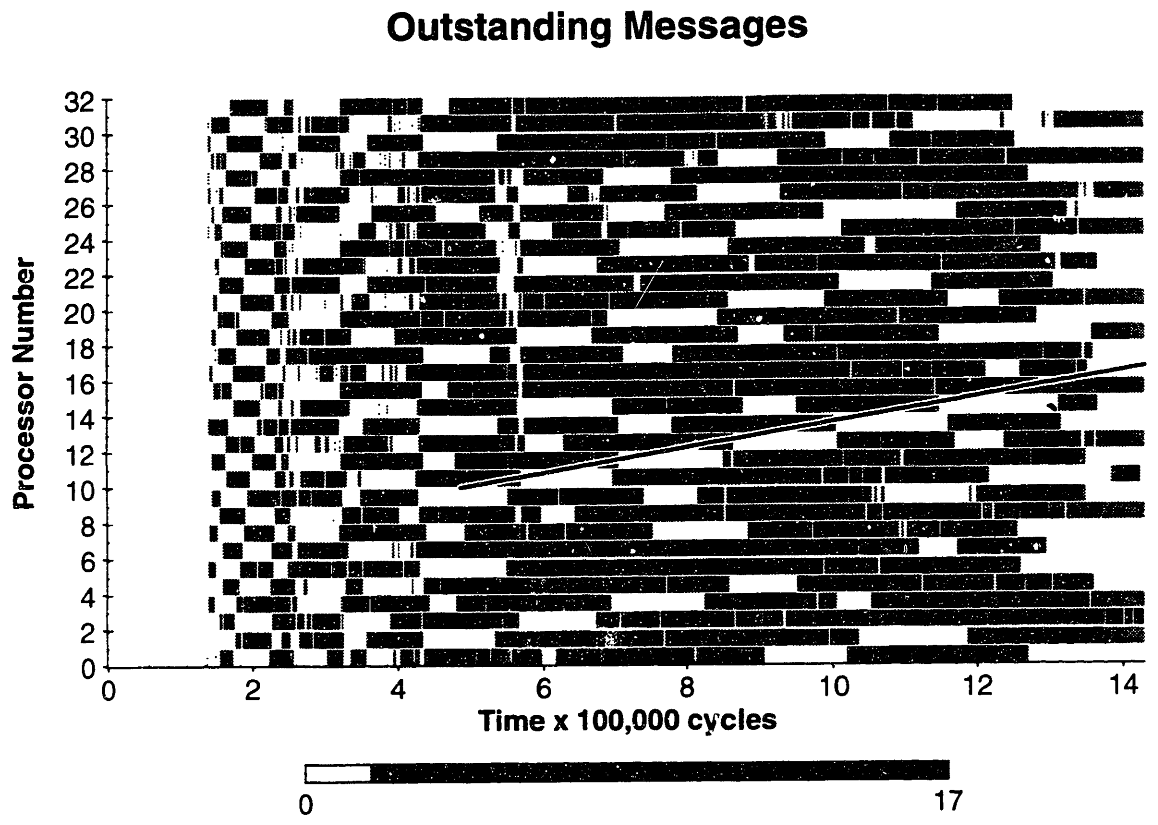


Figure 7-7: Congestion for Transpose without Barriers

The graph shows the total number of packets in the network destined for each processors as the transposition progresses. Time is measured in 33-megahertz clock cycles. Dark regions correspond to congested receivers, while white areas mean that there are no messages in the network for that processor. The overlaid line highlights the movement of congestion as discussed below.

black and white regions thus form lines that rise at an angle of about 20 degrees from horizontal; we have explicitly marked one of these lines.

Consecutive transfers incur collisions that can be avoided by the use of barriers to ensure that only one permutation runs at a time. Without barriers, hot spots start due to random variation, and then persist systematically, gradually becoming worse and worse. The barriers increase the performance by eliminating target collisions due to consecutive permutations. Thus, patterns composed from permutations separated by barriers *never* have target collisions.

As a side note, Figure 7-7 also highlights the power of Strata's integrated support for graphics. Other than the overlaid line, the entire graph was produced from a six-line graph specification and Strata's support for application-specific data collection. This graph is particularly challenging because the injection time is recorded at the sending processor, which requires a global view of data collection and a globally consistent measure of time. We are unaware of any other graphs in the literature (for real multiprocessors) with the resolution and multiple processor data collection represented here.

7.3.2 Analysis and Optimizations

We have explained, at least in part, the large differences between the versions with and without barriers. Now we return to Figure 7-6 to examine other anomalies of the graph. For the barrier-free versions, we expect the performance to drop monotonically as the block size increases, but for very large block sizes the performance actually increases. This is an artifact of fixing the total data transmitted rather than the number of cyclic shifts. For large block sizes we perform very few shifts, and with so few transitions from one permutation to the next, the system never gets far out of sync. To demonstrate that large blocks actually do show worse performance, we reran the experiment with 100 rounds instead of only 32:

Permutations	MB/sec	95% CI
32	2.22	±0.122
100	1.67	±0.040

The new data point, 1.67 megabytes per second, fits right on the asymptote implied by the rest of the "Strata Without Barriers" curve. Thus, without barriers, as the block size increases the performance approaches 1.67 megabytes per second for Strata. The asymptote for CMMD is 1.43 megabytes per second.

The performance of Strata with barriers drops slightly for very large transfers. This is due to cache effects: each sender sends the same block over and over. For all but the largest blocks, the entire block fits in the cache. The performance of CMMD with barriers does not appear to drop with the large blocks; in actuality it does drop but the effects are masked by the high variance of `CMAML_scopy`. The differences in performance and variance between Strata and CMMD are quite substantial; they are due primarily to bandwidth matching and reflect the difference in block-transfer performance as discussed in Section 6.1.3.

The versions with barriers perform worse for small blocks simply because of the overhead of the barrier, which is significant for very small block sizes. The "Optimized Barriers" curve shows an optimized version that uses fewer barriers for small transfers. The idea is to use a barrier every n transfers for small blocks, where n times the block size is relatively large, so that the barrier overhead is insignificant. The actual n used for blocks of size B is:

$$n = \left\lceil \frac{512}{B} \right\rceil \tag{52}$$

The choice of 512 is relatively unimportant; it limits barriers to about 1 for every 512 bytes transferred. Smaller limits add unneeded overhead, while larger limits allow too many transitions between barriers and risk congestion. (At $n = \infty$, there are no barriers at all.)

Another important feature of Figure 7-6 is that the barrier-free versions have a hump for 16-byte to 400-byte block sizes. The increase from 16 to 32 bytes is due to better amortization of the fixed start-up overhead of a block transfer. The real issue is why medium-sized blocks perform better than large blocks. This is because the number of packets sent in a row to one destination is relatively low; the packet count ranges from 2 to 8 for blocks of 32 to 128 bytes. With such low packet counts, the processor switches targets before the network can back up, since the network can hold about ten packets per node. Between receiving and switching targets, the time for a transition allows the network to catch up, thus largely preventing the "sender groups" that form with longer block transfers. Section 7.4 examines this effect in more detail.

Finally, we have seen some cases in which using barriers more frequently than just between rounds can improve performance. This occurs because barriers act as a form of global all-pairs flow control, limiting the injection rate of processors that get ahead. For very large machines, the bisection bandwidth becomes the limiting factor rather than the software overhead. In such cases, the extra barriers form a closed-loop flow-control system, while bandwidth matching alone is an open-loop mechanism. We have not yet determined the exact benefit of the extra barriers, but for machines with 128 processors or less, bandwidth matching seems to be sufficient by itself.

7.3.3 The Impact of Bandwidth Matching

Figure 7-8 shows the impact of bandwidth matching on sequences of permutations, in particular the transpose pattern. Without any delay, Strata actually performs worse than CMMD for medium to large blocks. This occurs precisely because Strata has lower overhead and thus a correspondingly higher injection rate than CMAML_scopy. Increasing the delay to 28 cycles, as shown in the last chapter, ensures that the injection rate is slightly slower than the reception rate. The added delay not only increases the performance by about 25%, it also reduces the standard deviation by about a factor of fifty! The drop in variance occurs because the system is self-synchronizing: any node that gets behind quickly catches up and resumes sending.

The poll-once version takes this a step farther. Given that everyone is sending at nearly the same rate, it is now sufficient to pull out only one packet, since it is unlikely that there will be two packets pending. Polling when there is no arrival wastes 10 cycles, which explains the 10% improvement in throughput. Unlike the case without bandwidth matching, the network remains uncongested even though less polling occurs. Optimum performance requires both bandwidth matching and limited polling. To put Strata's performance in perspective, the CM-5 study by Kwan *et al.* measured block-transfer performance at 10.4 megabytes per second per node [KTR93]; Strata sustains more bandwidth *for transpose*, at 10.66 megabytes per second, than they saw for individual messages. The net improvement over CMMD without barriers is about 390%, which is the difference actually seen by users when they switch from CMMD to Strata.

Although limited polling can improve performance, it is not very robust. When other limits such as the bisection bandwidth become bottlenecks, limited polling causes congestion. We expect that limited polling is appropriate exactly when the variance of the arrival rate is low; if the arrival rate is bursty (due to congestion), the receiver should expect to pull out more than one packet between sends. In practice, this means that patterns such as 2D-stencil that do not stress the bisection bandwidth can exploit limited polling, while random or unknown patterns should poll until there are no pending packets.

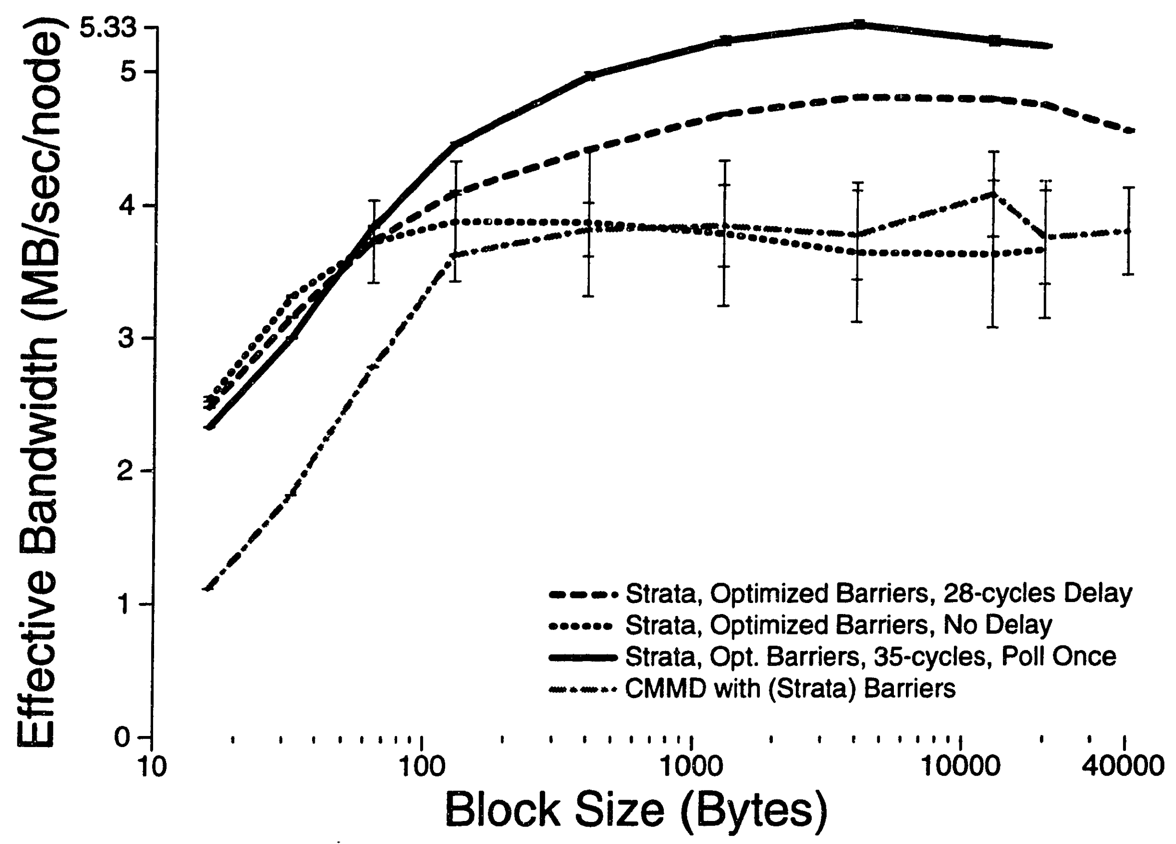


Figure 7-8: Bandwidth Matching and Transpose

The graph shows the impact of bandwidth matching on the transpose pattern. The upper two curves use bandwidth matching while the lower two do not. The poll-once version performs the best, actually reaching the block-transfer upper bound on the difficult transpose pattern. The bandwidth-matched versions perform 25-35% better and reduce the standard deviation by a factor of 50.

Introducing delay for short transfers actually hurts performance, as shown by the superior performance of the "No Delay" version for small transfers. In this case, the start-up overhead introduces plenty of delay by itself; any additional delay simply reduces the bandwidth. Thus, future versions of Strata will adjust the delay depending on the length of the transfer. For the limited polling case, it was slightly beneficial to increase the delay to 35 cycles to allow late processors to catch up more quickly. Note that the delay has no effect on the steady-state performance in which each processor simply alternates between sending and receiving.

As discussed in Section 6.1.3, bandwidth matching is essentially a static, open-loop form of flow control. However, despite its lack of feedback, bandwidth matching is quite stable due to its self-synchronizing behavior: the state in which each node alternates between injection and reception is stable. Traditional flow control via end-to-end acknowledgments would be more robust, but it is too expensive, since each acknowledgment requires significant overhead at both ends. A relatively cheap closed-loop mechanism for many situations is to use barriers as all-pairs end-to-end flow control. In some early experiments we found that frequent barriers improved performance; some of this effect occurred because barriers limit the injection rate, for which bandwidth matching is more effective. Nonetheless, we expect that frequent barriers may be a cost effective flow-control technique when the limiting factor is internal network bandwidth, rather than the software overhead.

7.4 Patterns with Unknown Target Distributions

The previous section showed that large increases in bandwidth could be achieved by using barriers to prevent interference between adjacent permutations. However, it is not always possible to set up the communication as a sequence of permutations. Instead of a sequence of permutations, the communication pattern becomes simply a sequence of *rounds*, in which the targets for each round may or may not form a permutation. In this section, we show that in cases where collisions may occur within a round but the overall distribution of targets is still uniform, interleaving packets is the key to performance. Interleaving packets is a novel technique that mixes packets from different block transfers, as opposed to standard approach of sending all of one transfer's packets followed by all of the next transfer's packets.

Figure 7-9 shows the effective bandwidth versus the size of the block sent to each target for a variety of Strata-based communication patterns. The "Random Targets" version picks each new target randomly from a uniform distribution. Thus, within a round we *expect* target collisions and barriers do not help. The key is that for small blocks the collisions do not matter because they are short lived; the sender switches targets before the onset of congestion due to target collisions. For large blocks, the collisions persist for a long time and thus back up the network, reaching the same asymptote as the "No Barriers" curve, which is the transpose pattern without barriers and thus also has a uniform distribution.

The key conclusion from this graph is that when the distribution is unknown, small batch sizes avoid prolonged hot spots. For example, if a node has 10 buffers that require 100 packets each, it is much better to switch buffers on every injection, a batch size of one, than to send the buffers in order, which implies a batch size of 100.

7.4.1 Asynchronous Block Transfers

To explore this hypothesis, we built an asynchronous block-transfer interface, which is given in Figure 7-10. Each call to the asynchronous block-transfer procedure, `AsyncBlockXfer`, sends a small part of the transfer and queues up the rest for later. After the application has initiated several transfers, it calls a second procedure, `ServiceAllTransfers`, that sends all of the queued messages. The second procedure sends two packets from each queued transfer, and continues round-robin until all of the transfers are complete. To avoid systematic congestion, the order of the interleaving is a pseudo-random permutation of the pending transfers. Thus, each processor sends in a fixed order, but different processors use different orders.

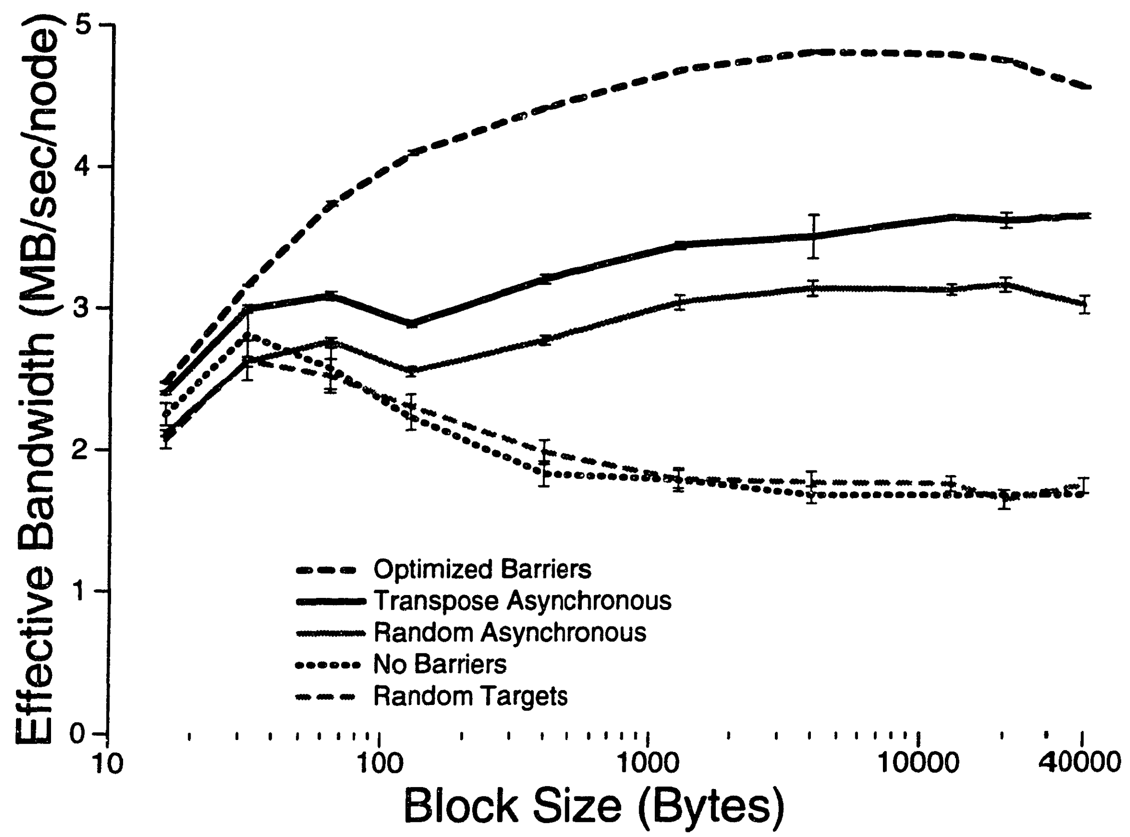


Figure 7-9: The Effect of Interleaving Packets

The two asynchronous block-transfer versions use packet interleaving to achieve about twice as much bandwidth as the corresponding synchronous block transfers. The version with barriers still performs much better, but it applies only when the communication can be structured as a sequence of permutations; the asynchronous block-transfer interface avoids this requirement.

7.4.2 Results for Packet Interleaving

The performance of this technique appears in Figure 7-9 as the two "Asynchronous" curves. For random targets, interleaving the packets increases the bandwidth by a factor of about 1.8 for large blocks (more than 400 bytes) and performs about the same for small transfers. When used for the transpose pattern, interleaving the packets increases the performance by a factor of 2.1 for large transfers, and by about 15% for small blocks. The transpose pattern performs better because the distribution is more uniform than random targets. The version with barriers still performs substantially better, but it requires global scheduling: each round *must* be a permutation. Thus, packet interleaving should be used when the exact distribution of targets is unknown. In general,

```
ABXid AsyncBlockXfer(int proc, unsigned
   port, int offset, Word *buffer, int
   size, void (*complete)(ABXid id, Word
   *buffer))

void ServiceAllTransfers(int rounds)

void CompleteTransfer(ABXid id)
```

Figure 7-10: Asynchronous Block-Transfer Interface

These three procedures implement Strata's asynchronous block transfer. The first routine, `AsyncBlockXfer`, sends part of the block and queues up the rest. `ServiceAllTransfers` completes all of the pending transfers by interleaving their packets. `CompleteTransfer` forces the completion of one particular transfer.

the difference between the asynchronous transfers and the version with barriers is due to the overhead for interleaving.

The dip at 128-byte transfers occurs because there is no congestion for smaller messages, and because the substantial overhead of the interface is amortized for larger messages. Using adaptive-delay bandwidth matching, as proposed in Section 7.3.3, should eliminate this dip by reducing the overhead for smaller transfers.

Packet interleaving allows the system to avoid *head-of-line blocking*, which occurs when packets are unnecessarily blocked because the packet at the head of the queue is waiting for resources that those behind it do not need. Karol *et al.* showed that head-of-line blocking can limit throughput severely in ATM networks [KHM87]. Although Strata tries to send two packets from each message at a time, if it fails it simply moves on to the next message. This has no effect on the CM-5, however, because the network interface contains a FIFO queue internally, which allows head-of-line blocking regardless of the injection order. Thus, we expect interleaving to be a bigger win on machines that combine deterministic routing with a topology that has multiple output directions, such as a mesh or torus. In general, *all* levels of the system should avoid head-of-line blocking; avoiding it in a only a subset of the hardware, operating system, libraries, and application provides no benefit!

We looked at a variety of interleaving strategies. The ideal order theoretically is to pick the next transfer randomly from those that remain incomplete, perhaps weighting the choices by how many packets remain, so that the transfers tend to complete at the same time. Unfortunately, generating a random number for every transmitted packet is prohibitively expensive and completely dominates any potential gain from interleaving. The strategy used by Strata is to order the transfers randomly at the time they are queued up (by `AsyncBlockXfer`). This strategy leads to a fixed round-robin order that differs for each processor. Although not as good theoretically as true random interleaving, this strategy only requires one random number per transfer and still greatly reduces the chance of systematic congestion. Although very unlikely, it is possible for this strategy to exhibit prolonged congestion if multiple processors pick the same order; the pure random version avoids this problem.[1] An even simpler strategy is to use plain FIFO round-robin ordering, but this costs almost as much as random ordering and is much less robust, since it is quite easy for multiple processors to use the same order.

We also looked at varying the number of transfers given to the module at one time. The curves in the figure use 8 at a time even though it is possible in this case initiate all 64 immediately (for one transpose). The curves for 4 and 16 look nearly the same, while 32 and 64 seem to do a few percent better on average. Using less than three transfers leads to markedly worse performance, but the actual results greatly depend on the selected ordering. Overall, the Strata interface seems quite robust, although it works best with at least four transfers at a time.

The benefit of interleaving has important consequences for message-passing libraries. In particular, any interface in which the library sends one buffer at a time is fundamentally broken. Such an interface prevents interleaving. Unfortunately, the one-buffer-at-a-time interface is standard for message-passing systems. To maximize performance, a library must allow the application to provide many buffers simultaneously.

7.5 Conclusions

With naive message-passing software and processor-network interfaces, a parallel computer may achieve only a fraction of the communications performance of the underlying network. Our experiments indicate that there are several mechanisms and techniques that can be used at various levels in the system to ensure high performance.

We found three underlying mechanisms that can improve performance. Barriers can be used to determine when all processors are finished sending, or when all are finished receiving. The order in which packets are injected into the network can be managed; we studied interleaving and randomized injection orders. We also demonstrated the importance of bandwidth matching to high-level communication.

We found several reasons why these mechanisms work. They help avoid target collisions, in which several processors are sending to one receiver at the same time. They also help to smooth out, over time, the bandwidth demands across various cuts of the network. These mechanisms also provide various forms of flow control, which improves the efficiency of the network. For example, barriers act as global all-pairs flow control, guaranteeing that no processor gets too far ahead at the expense of another processor.

The following rules-of-thumb can help programmers decide when and how to use each of these mechanisms:

1. If possible, recast the communication operation into a series of permutations. Sepa-

1. For both of the random interleavings, it is critical that the processors use different seeds! Otherwise, all of the processors will use the same order. Strata seeds its random-number generator, which is based on the "minimal standard" algorithm [PM88], using a function of the processor's number and the value of its cycle counter.

rate the permutations by barriers, and use a bandwidth-matched transfer routine, such as those provided by Strata, to implement each permutation. We found that this strategy can improve performance by up to 390%.

2. If bandwidth matching is impractical, because, for example, the real bottleneck is some internal cut of the network, then using periodic barriers inside each permutation may help. We have seen cases where barriers within a permutation improve performance.
3. If you know nothing about the communication pattern, you should use an interleaved or randomized injection order, as provided by Strata's asynchronous block-transfer mechanism. Even in this case, periodic barriers within the transfers may improve performance.
4. It is important to keep the network empty: it is almost always better to make progress on receiving than on sending. Thus, the best default behavior is polling until there are no pending arrivals. The one exception occurs when the variance of the arrival rate is near zero, which can occur with bandwidth matching, in which polling exactly once is sufficient and actually increases the effective bandwidth.
5. If the computation operation consists of two operations that have good performance separately, then keep them separate with a barrier. It is difficult to overlap communication and computation on the CM-5 because the processor must manipulate every packet, and the low capacity and message latency of the CM-5 network reduce the potential gain from such overlap. However, very large block transfers interact poorly with the cache. We have seen cases where limited interleaving of the communication and computation can increase the effective bandwidth by about 5%, because the transferred data, which was just computed, remains in the cache.

These guidelines form the basis of Strata's transpose and asynchronous block-transfer modules. They extend the performance of Strata's block transfers to complex communication patterns, which increases performance by up to a factor of four.

7.5.1 Implications for Network Design

Our results indicate that it may be a good idea to place some of these mechanisms into the network and the processor-network interface. A parallel computer should provide a rich collection of global flow-control mechanisms. Almost any form of flow control is better than none, but the best choice is not clear. Bandwidth matching is a software-only technique with near zero overhead, but it is an open-loop mechanism that requires an over-engineered network to remain robust. Hardware end-to-end flow control is far more robust but is complex and expensive. Closed-loop software mechanisms such as round-trip acknowledgments and barriers require less hardware support, but still have significant overhead.

Communication coprocessors, which seem to be on the rise — Cray's T3D, MIT's *T, Stanford's FLASH, Wisconsin's Typhoon, MIT's Alewife, and Intel's Paragon — increase the need for flow control by augmenting the bandwidth achievable by the sender. At the same time, a coprocessor without significant programmability may reduce or eliminate the power of software techniques such as bandwidth matching and interleaving. Thus, a successful coprocessor will probably involve either hardware bandwidth matching or true end-to-end flow control and some form of concurrent block transfers.

A parallel computer should support fast predictable barriers. The cost of a barrier should be competitive with the cost of sending a message. Perhaps more importantly, the behavior of barriers should be independent of the primary traffic injected into the data network. The CM-5 provides such barriers by using a hardware global-synchronization network; multiple priorities or logical networks would also be sufficient.

The receiver should be at least as fast at the sender. Allowing user-level access to the network interface is the most important step in this direction. However, hardware support to speed up the receiving of messages even by a few cycles would improve both the performance and predictability of CM-5 applications, especially for those applications with dynamic communication patterns that are not amenable to global scheduling via barriers.

The network, the processor-network interface, and its software should provide mechanisms to manage the order in which packets are injected into the network. A communication coprocessor for sending and receiving packets, such as those proposed for MIT's *T [PBGB93] and Stanford's FLASH [Kus+94] machines, can make it easier to overlap communication and computation, but our experiments indicate that such engines will require fairly sophisticated packet-ordering policies in addition to the aforementioned support for flow control. Similarly, very large packets are probably a bad choice because they may prevent effective packet reordering. The *entire* system must avoid head-of-line blocking.[1]

7.5.2 Relations to Network Theory

Our experiments provide empirical evidence that some of the strategies used by theorists to prove theorems also make sense in real parallel systems. Theorists have argued that slowing things down can speed things up [FRU92] or provide predictable behavior [GL89]. [expand] We found that both barriers and bandwidth matching, which at some level slow down the system, actually speed things up. The use of barriers prevents processors that have gotten a little bit ahead from widening their lead. Bandwidth matching is analogous to freeway on-ramp meters, which limit "injection" to reduce the variance of the arrival rate, which in turn keeps traffic flowing smoothly. Other examples of slowing things down to keep them working well include Ethernet's exponential backoff [MB76], and the telephone system's approach to dealing with busy calls by forcing the caller to redial rather than wait in a queue (page 103 in [Kle76]).

Not only are there sound theoretical arguments for randomized injection order [GL89], but we found significant practical application for reordering, even when we started with what we thought was a reasonable injection order, such as randomly selected targets.

1. Due in part to this research, the next generation communication coprocessor from Thinking Machines Corporation is very likely to provide hardware support for both end-to-end flow control and packet interleaving.

Theorists have also argued that measuring the load across cuts of a network is a good way to model performance and to design algorithms [LM88]; we have seen a few situations where we could apply such reasoning to make common patterns such as transpose run faster.

7.5.3 Summary

We believe that our results apply to a wide range of parallel computers because the effects we observed are fundamental. Bandwidth considerations, scheduling issues, flow control, and composition properties will appear in any high-performance communications network. In particular, the rate at which a receiver can remove messages from the network may be the fundamental limiting issue in any reasonably engineered network.

Finally, it is worth repeating that users of Strata obtain the benefits of these mechanisms automatically, since it incorporates these techniques into the modules that implement high-level communication patterns such as transpose.

7.6 Related Work

Flow control has been an important part of long-distance networks for many years [Tan88]. The use of window flow control and sequence numbers is well known. For these networks, the overhead of acknowledgments is insignificant compared to other overheads such as the time of flight and the kernel-based processing at both ends. High-bandwidth networks, such as those for suitable for video, require precise flow control to avoid swamping the receiver with too much data. They typically start with a relatively low flow and incrementally increase it until the system starts to drop packets [Jac88]. There are a large variety of mechanisms for adjusting the window size [PTH90], including some that are self-optimizing [LM93]. These networks *require* a closed-loop system. Bandwidth matching also leads to an agreed-upon rate, except that the rate is statically determined and thus forms an open-loop system. We trade the robustness of a closed-loop mechanism, such as window flow control, for the zero overhead of bandwidth matching.

The AN2 ATM network from DEC [AOST93] provides hardware flow control for each link. This reduces the software overhead substantially and is part of the general trend toward low-latency communication on local-area networks. Again, flow control is *expected* for these networks.

In contrast, multiprocessor networks traditionally have provided no support for flow control other than congestion. These networks are not allowed to drop packets and thus the supporting software can avoid sequence numbers and acknowledgments entirely.[1] The flow control provided by bandwidth matching and barriers attacks the swamping issue associated with local-area and long-distance networks. The key differences are the very low overhead of Strata's mechanisms, and that fact that Strata can assume that all of the packets arrive correctly.

1. The CM-5 essentially requires sequence numbers for block transfers, because it does not preserve the order of messages. However, the normal implementation hides the sequence number by explicitly adding it into the receiver's base address for the block. This is sufficient for block transfers, since the order of arrival is unimportant: it only matters when the last packet arrives.

As mentioned in the main text, the use of barriers between permutations was first exploited by Steve Heller of Thinking Machines Corporation. The use of barriers to speed up all-pairs communication for an FFT application was noted by Culler *et al.* [CKP+93], but not investigated further. Our work explains the phenomenon and provides the split-phase barriers required to exploit it reliably without the overhead of interrupts.

Another effort to speed up all-pairs communication was performed by the iWarp group at CMU. Their technique [HKO+94] involves hardware modifications to enforce the phases in the switches, which leads to optimal scheduling of the block transfers for the iWarp processor [PSW91]. We use barriers to enforce the phases, which although perhaps not be as scalable, are available on several commercial architectures.

There are several machines that support communication either on chip or via coprocessors, including MIT's Alewife [Aga+91], MIT's *T [PBGB93], MIT's J-machine [Dal+89], the Cray T3D [Ada93][Cray92], Intel's Paragon [Intel91], Stanford's FLASH [Kus+94], Wisconsin's Typhoon [RLW94], and CMU's iWarp [Kun90][PSW91]. None of these processors support hardware flow control, but all of them ensure correct delivery. Of these, Alewife is partially programmable, FLASH and *T are programmable by the operating-system designers, Typhoon is user programmable, and the others are essentially not programmable. Bandwidth matching and packet interleaving are probably beyond all but Typhoon, FLASH, *T, and perhaps Alewife. Alewife, iWarp, and the T3D provide statically matched transmission rates for DMA block transfers, but not for communication in general. Alewife could support packet interleaving in software on the primary processor, but would forfeit the performance advantage of the coprocessor's DMA transfers.

Section 7.5.2 covers the related theoretical work.

8

Extensions and Future Work

This chapter looks at extensions and other uses of the statistical-modeling technology developed in this dissertation. We look at three areas of future work in particular:

1. Model factoring, which is the use of hierarchical models,
2. Input-dependent performance, which occurs when an inplementation's performance is strongly dependent on the particular input, rather than just the size of the input, and
3. Real high-level libraries, which require more than the just the basic technology investigated so far.

In looking at these areas, we hope to both hint at what is possible and provide insight on the key issues for making these techniques a common technology.

8.1 Model Factoring

Model factoring was touched on in the discussion of parameter optimization for the stencil module. The basic idea is to model an application in terms of the models for the run-time system, rather than directly via a list of terms. The resulting model reflects the application's use of the run-time system. For example, consider the factored model from the stencil parameter optimization:

$$\text{Communication}\,(\mathit{width}) = \mathit{StencilHeight} \cdot \text{BlockTransfer}\left(\frac{8 \cdot \mathit{width}}{2\,(8)}\right) \tag{53}$$

where:

width ≡ Width of the virtual grid

StencilHeight ≡ Number of rows along the top and bottom boundary that must be transmitted vertically (normally 1)

BlockTransfer ≡ The cost of sending a block transfer

This model is not a linear combination of terms and was not generated by the auto-calibration toolkit. Instead it was built by hand based on the number of calls to Strata's block-transfer routine and the sizes of the blocks for those calls. This model has two key advantages:

- ❑ The model never has to be recalibrated, it always reflects the current environment. We have hidden the platform dependencies in the submodel for block transfer. Thus, as soon as the Strata models are calibrated, all of the factored models are ready to use.
- ❑ The bigger advantage is that factored models capture the relationship between the run-time system and the application. Thus, if we could estimate the performance of Strata on the new platform, which is relatively easy because Strata is low level, then the factored models allow us to predict application performance using only the Strata models.

Factored models also have some disadvantages:

- ❑ They are fundamentally less accurate. For example, the above model ignores the variation in loop overhead and cache effects. This works out reasonably well for this particular model, since the block-transfer costs dominate the other effects. By moving the platform dependencies inside the submodel, we throw out dependencies that arise during the composition of submodels, such as loop overhead. Section 8.1.3 discusses hybrid models that attempt to capture such dependencies while retaining the hierarchical structure.
- ❑ Factored models are harder to build. For regular models, the designer determines only the list of relevant terms and the toolkit does the rest. For factored models, the designer must count operations carefully and then express the performance as a combination of the available submodels. The difficulty of this process is also much of the benefit: capturing this structure is the key information that supports the techniques covered in the next few subsections.

The relative accuracy of factored models to pure statistical models is unclear and very application dependent. We expect that most uses of the statistical models could use factored models instead, but with some penalty in terms of accuracy. The one serious use of a factored model, parameter optimization for stencils, seemed unaffected by the reduced accuracy.

8.1.1 Impact Reports and Cross Prediction

A fairly straightforward use of factored models is the investigation of the impact of changes to the architecture, operating system, or Strata run-time system. For example, if we wanted to know the impact of doubling the block-transfer throughput, then we would replace the block-transfer model with one that reflects the expected performance after the change. Given factored models for applications, we can then predict the impact of the change on *application* performance. The key point here is that factored models allow us to leverage knowledge about Strata, which is relatively easy to predict, into information about application performance, which is very hard to predict.

An extension of this idea is *cross prediction*, in which we estimate the Strata models for a proposed architecture, and then leverage the Strata models into predicted application performance for that architecture. This could be used, for example, to estimate the usefulness of a new machine without having to port applications to it. Although imperfect, this kind of analysis would be far more accurate than estimating application performance directly from the machine specifications.

Taken a step further, we could build a *model-benchmark suite*, which consists of the factored models for several important applications. Given this set, we could quickly evaluate the performance of a machine simply by estimating the performance of the run-time system. It remains to be seen if this is a viable technique, but it could provide a significant level of accuracy with relatively little work. Again, the hard part of the prediction is captured by the factored models.

8.1.2 Symbolic Differentiation

A similar technique allows us to determine which coefficients of the Strata models have the most impact on an application. Given a factored model and a particular set of Strata models, we can determine the exact impact of any of the Strata coefficients on the overall application performance. The idea is the take the symbolic derivative of the factored model with respect to the coefficient of interest. This can be done using a symbolic-math package such as Mathematica [Wol91].

The result is an expression that computes the rate of change in application performance as we change the chosen coefficient. If we repeat this process for all of the coefficients (automatically), then we can sort the terms by their impact on the application. For example, we might find that the most important Strata coefficient for radix sort is the start-up cost for vector scan operations, which are used to compute the global counts. This technique would provide a novel form of profiling information that could be used to tune Strata in the most productive manner: we would optimize those items that have the most impact on application performance.

8.1.3 Hybrid Models

Finally, it is possible to combine factored models with statistical modeling, by treating the submodels as terms. For example, the hybrid version of the factored stencil model is:

$$\text{Communication}\,(\mathit{width}) = \alpha + \beta \cdot \mathit{StencilHeight} \cdot \text{BlockTransfer}\left(\frac{8 \cdot \mathit{width}}{2\,(8)}\right). \qquad (54)$$

The coefficients, α and β, can be set by the auto-calibration toolkit. These coefficients would capture most of the composition effects such as loop overhead and cache effects. Thus, hybrid models are more accurate than regular factored models, but probably slightly less accurate than pure statistical models, since they have fewer degrees of freedom. The hybrid model must be recalibrated to maintain its accuracy, but it still captures the relationship between the application and the run-time system. If used for cross prediction, the hybrid coefficients would simply be left at their values for some calibrated platform. Although these coefficients are probably not perfect for the new platform, they should do better than the pure factored models, which completely ignore the composition costs.

Thus, factored models and hybrid models represent a different approach to application modelling. The emphasis is on structure rather than predictive power. The structure enables new abilities including evaluation of architectural changes and proposed platforms, and model-based profiling for predicting the run-time system modules with the most potential for improving application performance.

8.2 Capturing Input Dependencies

For all of the applications discussed so far, the performance of the application either did not depend on the input, on depended only on its size. There are applications, however, whose performance depends on the specific input. For example, sparse-matrix applications completely depend on the structure of the matrix. Some matrices contain plenty of parallelism, while others may have sequential dependency chains. Thus, it is impossible to predict the performance of the application without seriously looking at the input. In fact, for sparse matrices, essentially all of the input affects the performance: there is no subset or summary statistic with predictive power.

This example is actually the worst case, so there may be some hope for capturing input dependencies. For example, there are accurate B-tree models that depend on the percentage of update operations in the total workload [Cos94]. This summary statistic is sufficient to predict the throughput of the B-tree, even though it does not capture the resulting structure of the tree, which is totally dependent on the input. Other applications may also have summary statistics that are sufficient for prediction, such as information on the distribution of the input.

Another approach is possible as well. Although it is essentially impossible to predict the performance of a sparse-matrix solver without looking at all of the input, it possible to make predictions after the matrix has been partitioned among the nodes, which is the first step of the solver [CSBS94]. By looking as the partitioning of the matrix, we can determine the amount of communication required and the degree of load balancing. Thus, while prediction straight from the input seems hopeless, prediction after the partitioning is relatively easy and would allow algorithm selection or parameter optimization for the remainder of the application.

8.3 Real High-Level Libraries

The applications in this dissertation are intended to prototype the technology, rather than to provide production-quality libraries. Obviously, it would be great to build real high-level libraries that deliver portability with high performance. The primary obstacle is simply getting good implementations that are fast, robust, and numerically stable. We hope to build some real libraries in the near future with the help of application specialists to ensure that the libraries are useful to computational scientists.

There has been strong interest in building such a library, primarily because people are tired of rewriting their applications. The expectation is that a high-level library is worth extra effort because it is more likely to remain viable over time, and because it supports the incorporation of new implementations from third parties.

9

Summary and Conclusions

This work demonstrated the promise of automatic parameter selection and parameter optimization, which is portability with high performance. We achieve better performance because we use the right algorithm for the target platform and workload, and because the parameters for that algorithm are set optimally as well as automatically.

The key technology behind these portable optimizations is automatically calibrated statistical models, which precisely capture the behavior of each implementation. Moving to a new environment simply requires recalibrating the models so that the algorithm selector and parameter optimizer continue to make the correct decisions. Overall, the selector picked the best implementation more than 99% of the time, while the optimizer found the best parameter value 100% of the time, even though the platforms varied widely and required different algorithms and parameter settings.

To support statistical modeling, we developed the auto-calibration toolkit, which turns simple model specifications into complete, calibrated and verified models. Given the number of models involved in this thesis, the toolkit was priceless. It also led to a higher degree of confidence in the models, since building the models by hand is a long and error-prone process.

We also developed a high-performance run-time system, Strata, that supports the module implementations and the data collection required for statistical modeling. Strata and the high-level communication modules that it includes provide exceptional performance and predictability. In the process, we developed several mechanisms for improving communication performance including better active messages, bandwidth matching, packet interleaving, and split-phase synchronization operations.

9.1 Conclusions

Overall, this work met its objectives: we developed the technology required for automatic selection and optimization based on statistical models, and demonstrated that this technology could lead to libraries that can adapt to the platform and workload with no guidance from the end user. Although not production quality, the libraries in this work attack all of the significant problems, which includes automating the model-building process and then using the models for both selection and optimization. The following subsections summarize our conclusions for each of the major aspects of the system.

9.1.1 Automatic Algorithm Selection

- ❑ The models for both modules were quite accurate: the selector picked the best implementation more than 99% of the time on all of the platforms.
- ❑ In the few cases in which a suboptimal implementation was selected, that implementation was nearly as good as the best choice: only a few percent slower on average for the stencils and nearly identical for sorting.
- ❑ We showed that the system only makes errors near the intersections of the performance surfaces, from which it follows that the penalty for these errors is small.
- ❑ The benefit of picking the right implementation was often very significant: averaging 8-90% for stencils and often more than a factor of two for sorting.
- ❑ An important benefit of automatic selection is the tremendous simplification of the implementations that follows from the ability to ignore painful parts of the input range. By adding preconditions into the models, we ensure that the selector never picks inappropriate implementations. Thus, an immediate consequence of automatic selection is the ability to combine several simple algorithms that only implement part of the input range.
- ❑ The selection can be done at run time as well as compile time. The code expansion is small considering that all of the versions must be linked in, and the overhead of evaluating the models is insignificant compared to the execution time of the module. Run-time selection provides a module that achieves the performance of the best available implementation for the given workload. Unlike traditional hybrid algorithms, the selection mechanism is tuned the current environment and can ported with only a simple recalibration of the models.

9.1.2 Parameter Optimization

- ❑ Although there were a few errors in prediction for algorithm selection, we found no errors in the prediction of the optimal parameter value. We looked at 80 trials of each of the two stencil parameters, and 40 trials of the digit-width parameter for radix sort.

- ❑ In some ways parameter optimization is easier than algorithm selection: in particular, there is more room for error, since small changes in the parameter tend to lead to rather large changes in performance. Put another way, it is harder for two points on the same curve to be close for a discrete parameter, than it is for two different curves to be close.
- ❑ Model-based parameter optimization provides a form of adaptive algorithm that remains portable. In fact, one way to view this technique is as a method to turn an algorithm with parameters that are difficult to set well into adaptive algorithms that compute the best value. As with algorithm selection, this brings a new level of performance to portable applications: the key performance parameters are set optimally and automatically as the environment changes.
- ❑ Run-time parameter optimization has low overhead, no code expansion, and can exploit workload information, such as the problem size, that is only available dynamically. Thus, unless all of the factors are known statically, it is better to postpone parameter optimization until run time.
- ❑ Parameter optimization can be combined with automatic algorithm selection to produce simpler, more powerful modules, with better overall performance. The selection is done in two passes: first, we find the optimal parameter values for each of the parameterized implementations, and then we select among the optimized versions.

9.1.3 The Auto-Calibration Toolkit

- ❑ The model specifications are simple and tend to be quite short, usually about one line per model.
- ❑ The toolkit removes irrelevant basis functions automatically. This is critical because it allows the designer to add any terms that he believes *might* be relevant.
- ❑ When there are missing basis functions, indicated by an inaccurate model, the toolkit reveals which of the terms are relevant and generates the residuals, which can be used to identify the missing basis functions.
- ❑ The code for sample collection is generated and executed automatically, thus eliminating one of the more painful steps of statistical modeling.
- ❑ Sample collection is robust to measurement errors, which eliminates another painful aspect of statistical modeling. In particular, the output of the profiling code is a sequence of reliable samples.
- ❑ The tools support minimization of the relative error in addition to the traditional minimization of absolute error. For performance models, minimizing relative error leads to better models. We also defined the mean relative error as a more useful metric of the accuracy of a model.
- ❑ The toolkit tests the produced model against independent samples to ensure that

the model has good predictive power. During recalibration (for a new architecture or a change in system software) usually all of the models pass verification, which means that the amount of human intervention required for recalibration is essentially zero.

- ❑ Finally, the toolkit produces C and perl versions of the models that can be embedded within a larger system, such as tools for algorithm selection and parameter optimization.

9.1.4 Porting

- ❑ In general, porting a high-level library requires much less work than porting an application. The implementations depend on only C and Strata, and thus are trivial to port once those two pieces are available. Unlike traditional applications, high-level libraries require little if any tuning for the new platform. The libraries are essentially self-tuning: they combine the flexibility of multiple implementations with an automatic method for selecting the best one in the new environment. As long as at least one implementation runs well on the new platform, the library as a whole achieves good performance.
- ❑ Applications that use the libraries are also more portable, because many of the performance-critical decisions are embedded within the libraries and thus can be changed without affecting the application source code.

9.1.5 The Strata Run-Time System

- ❑ Strata provides a fast, safe split-phase interface for synchronous global communication, with many operations and very general segmentation options. The split-phase versions allow complicated protocols without deadlock, which includes mixing global operations with message passing.
- ❑ Strata's active messages are extremely fast: we obtained speedups of 1.5 to 2 times for a realistic sparse-matrix application. Strata's active-message performance comes from efficient implementation, effective use of both sides of the network, and flexible control over polling.
- ❑ The block-transfer functions achieve the optimal transfer rate on the CM-5, reaching the upper bound set by the active-message overhead. The key to sustaining this bandwidth is a novel technique called bandwidth matching that acts as a static form of flow control with essentially zero overhead.
- ❑ Bandwidth matching increases performance by about 25% and reduces the standard deviation for the performance of complex communication patterns by a factor of fifty.
- ❑ Strata provides timing and statistical operations that support the data collection required for automatic model generation.
- ❑ Strata's support for development includes printing from handlers, atomic

logging, and integrated support for state graphs that provide insight into the global behavior of a program.

- ❑ By using PROTEUS as the primary development platform, Strata users can exploit the mature tools of their workstation as well as the repeatability and nonintrusive monitoring and debugging provided by PROTEUS. This environment greatly reduced the development time for the high-level library applications.
- ❑ All of the Strata primitives have accurate performance models that were generated in whole or in part by the auto-calibration toolkit. These models lead to better use of Strata as well as to improvements in the Strata implementations.
- ❑ Strata is extremely predictable, which leads to more predictable applications and thus improves the power of statistical modelling for algorithm selection and parameter optimization.

9.1.6 High-Level Communication

- ❑ We developed three mechanism that lead to successful high-level communication: the use of barriers, the use of packet interleaving, and the use of bandwidth matching.
- ❑ Whenever possible, communication patterns should be composed from a sequence of permutations separated by barriers. We found that this strategy can improve performance by up to 390%.
- ❑ If you know nothing about the communication pattern, you should use an interleaved or randomized injection order, as provided by Strata's asynchronous block-transfer mechanism.
- ❑ It is important to keep the network empty: it is almost always better to make progress on receiving than on sending. Thus, the best default behavior is to poll until there are no pending arrivals.
- ❑ If the computation operation consists of two operations that have good performance separately, then keep them separate with a barrier. It is difficult to overlap communication and computation on the CM-5 because the processor must manipulate every packet, and the low capacity and message latency of the CM-5 network reduce the potential gain from such overlap.
- ❑ Very large block transfers interact poorly with the cache: it is better to alternate communication and computation at a granularity that keeps the computed information in the cache until the block transfer occurs. The benefit is about a 5% increase in throughput.
- ❑ Bandwidth matching leads to excellent performance for permutations. By providing flow control, bandwidth matching completely eliminates congestion for complex patterns such as transpose, which increases performance by at least 25% and predictability by a factor of fifty (in terms of the standard deviation).

- ❑ A successful coprocessor will probably involve either hardware bandwidth matching or true end-to-end flow control and some form of concurrent block transfers. Both of these features are required for good performance for complex patterns.
- ❑ The receiver should be at least as fast at the sender. Allowing user-level access to the network interface is the most important step in this direction. However, hardware support to speed up the receiving of messages even by a few cycles would improve both the performance and predictability of CM-5 applications, especially for those applications with dynamic communication patterns that are not amenable to global scheduling via barriers.
- ❑ The network, the processor-network interface, and its software should provide mechanisms to manage the order in which packets are injected into the network. The *entire* system must avoid head-of-line blocking.
- ❑ Both bandwidth matching and packet interleaving were brought about to some degree from the ability to capture and evaluate performance information quickly via the tools developed for this work. The insight that led to bandwidth matching came in part because the models pointedly revealed the relative costs of sending and receiving for both Strata and Thinking Machine's CMAML.

9.2 Limitations

There are three key limitations to this work. First, the models do not deal with applications whose performance is strongly tied to the particular input, such as a sparse-matrix solver. Section 8.2 discusses techniques that may be able to capture input dependencies successfully.

Second, we assume a sequence of phases, as in the SPMD programming model. The independence of the phases allows easy composition of the models, improves performance by enabling management of shared resources such as the network, and simplifies development by allowing phases to be debugged independently. This limitation could be relaxed in a couple of ways: the fundamental restriction is the composition of models. For example, we could partition the machine into groups of nodes and run different programs in each group. The models for the groups could be composed if the groups use different resources so that they do not interfere. To be more concrete, we could run independent programs in each of the four major subtrees of the CM-5; since the fat-tree allows independent partitions (along this particular cut), the programs would not interact and the time for the set of four would simply be the maximum of the group times.

The third major limitation is the assumption of a predictable architecture, which is only valid for *most* machines. For example, the Cray C90's performance is strongly tied to the access stride for arrays. It is not uncommon to see an order of magnitude performance improvement when a user changes the size of an array from a power of two to have *one* less element. Most users find this very irritating. The short answer is the that models simply do not apply to machines that are fundamentally unpredictable. For all concerned, I hope such severe anomalies disappear in the future, but if not, the models could perhaps be made to consider array layout, as least for the most important arrays. Of course, this anomaly also implies that the benefit of a clever parameter optimization would be an order of magnitude.

9.3 Closing Remarks

Although the specific conclusions have been covered in detail, there are a few "big picture" conclusions that can only come at the very end. It is hard to overstate the importance of the integration of the key components: the toolkit, Strata, high-level communication, and the model-based decision tools. Each of the components made the others better. Strata and the high-level communication modules made the applications much more predictable. Strata provides the data collection tools required by the toolkit, and the toolkit builds models that lead to the improvement of Strata and to the development of novel mechanisms such as bandwidth matching and packet interleaving. It is not an accident that I had these insights: I had better tools and better information about the behavior of the system. Bandwidth matching is obvious in hindsight, but was undiscovered for several years even though all of the symptoms were readily apparent.

Finally, I chose this topic because I believe that computer scientists have developed too many mechanisms, but not enough guidance or tools to select among them. I hope that this work leads to systems that can incorporate new mechanisms and algorithms and employ them in exactly the right situations, even if those involved know not the nature of those situations.

Bibliography

[Ada93]
D. Adams. *Cray T3D System Architecture Overview*, Technical Report, Cray Research, Revision 1.C. September 1993.

[Aga+91]
A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared-Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[And+92]
E. Anderson *et al. LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1992.

[ABC+88]
F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, Volume 5, Number 5, pages 617–640, October 1988.

[ACM88]
Arvind, D. Culler and G. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *International Journal of Supercomputer Applications*, Volume 2, Number 3, 1988.

[Ans73]
F. J. Anscombe. Graphs in Statistical Analysis. *The American Statistician*, Volume 27, pages 17–21, 1973.

[AOST93]
T. Anderson, S. Owicki, J. Saxe, and C. Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Transactions on Computer Systems*, Volume 11, Number 4, pages 319–352, November 1993.

[BBC+94]
R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics, 1994.

[BBD+87]
J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Reinhart and Winston, 1987.

[BBK94]
E. A. Brewer, R. Blumofe, and B. C. Kuszmaul. *Software Techniques for High-Performance Communication.* In preparation, 1994.

[BC90]
G. E. Blelloch and S. Chatterjee. VCODE: A Data-Parallel Intermediate Language. In the *Proceedings of Frontiers of Massively Parallel Computation*, October 1990.

[Ber91]
C. Bernadin. *The C/Math Toolchest.* Mix Software, Inc. and Pete Bernadin Software, 1991.

[BD92]
E. A. Brewer and C. N. Dellarocas. *PROTEUS User Documentation, Version 0.6.* Massachusetts Institute of Technology, November 1993.

[BDCW92]
E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. In *Proceedings of the ACM SIGMETRICS and Performance '92 Conference.* Newport, Rhode Island, June 1992.

[BCH+93]
G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. *CVL: A C Vector Library.* Carnegie Mellon University Technical Report CMU-CS-93-114, February 1993.

[BCH+94]
G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, M. Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, Volume 21, Number 1, pages 4–14, April 1994.

[BK94]
E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium (IPPS '94)*, Cancún, Mexico, April 1994.

[Ble89]
G. E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, Volume 38, Number 11, pages 1526–1538, November 1989.

[Ble90]
G. E. Blelloch. *Vector Models for Data-Parallel Computing.* Ph.D. Dissertation, MIT Press, 1990.

[BLM+92]
G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. *A Comparison of Sorting Algorithms for the Connection Machine CM-2.* Technical Report 222, Thinking Machines Corporation, 1992.

[BW93]
E. A. Brewer and W. E. Weihl. Development of Parallel Applications Using High-Performance Simulation. In *Proceedings of the Third ONR Workshop on Parallel and Distributed Debugging (WPDD '93)*, May 1993.

[CA94]
D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In the *Proceedings of the 1994 International Symposium on Computer Architecture (ISCA '94)*, pages 314–324, April 1994.

[CCL88]
M. Chen, Y. I. Choo, and J. Li. Compiling Parallel Programs by Optimizing Performance. *Journal of Supercomputing*, Volume 2, Number 2, pages 171–207, October 1988.

[CKP+93]
D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of Principles and Practice of Parallel Processing (PPoPP '93)*, May 1993.

[CLR90]
T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, MIT Press and McGraw-Hill, 1990.

[Cos94]
P. R. Cosway. *Replication Control in Distributed B-Trees.* Master's Thesis. Massachusetts Institute of Technology. September 1994.

[Cray92]
Cray Research. *Cray T3D Software Overview Technical Notes.* Cray Research Document Number SN-2505. Orders Desk: (612) 683-5907.

[Cray94]
Cray Research. *The Cray T3D Emulator Manual.* Cray Research Document Number SG-2500. Orders Desk: (612) 683-5907.

[CSBS94]
F. T. Chong, S. Sharma, E. A. Brewer, and J. Saltz. *Multiprocessor Run-Time Support for Fine-Grain Irregular DAGs.* Submitted for publication, 1994.

[Cyt85]
R. Cytron. Useful Parallelism in a Multiprocessor Environment. In the *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[Dal+89]
W. J. Dally *et al.* The J-Machine: A Fine-Grain Concurrent Computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[DBMS79]
J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK Users' Guide.* SIAM Publishing, 1979.

[DCDH88]
J. J. Dongarra, J. Du Cruz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms.* Argonne National Laboratory, Mathematics and Computer Science Division, August 1988.

[Ede91]
A. Edelman. Optimal Matrix Transposition and Bit Reversal on Hypercubes: All-to-All Personalized Communication. *Journal of Parallel and Distributed Computing*, Volume 11, pages 328–331, 1991.

[FJL+88]
G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems.* Prentice Hall, 1988.

[FLR92]
J. Feldman, C. C. Lim and T. Rauber. The Shared-Memory Language pSather on a Distributed-Memory Multiprocessor. In the *Proceedings of the Second Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory-Multiprocessors*, Boulder, Colorado, October 1992. Also *ACM SIGPLAN Notices*, Volume 28, Number 1, pages 17–20, January 1993.

[FORE92]
FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface: User's Manual.* 1992.

[FRU92]
S. Felperin, P. Raghavan and E. Upfal. A Theory of Wormhole Routing in Parallel Computers. In the *Proceedings of the 33rd Symposium on the Foundations of Computer Science (FOCS '92)*, pages 563–572, October 1992.

[FSF93]
Free Software Foundation. *Using and Porting Gnu CC, Version 2.0.* April 1993.

[GH93]
W. Gander and J. Hrebicek. *Solving Problems in Scientific Computing Using MAPLE and MATLAB.* Springer-Verlag, 1993.

[GJMW91]

K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. *Performance Prediction for Parallel Numerical Algorithms*. Technical Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[GKP89]

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*, Third Printing. Addison-Wesley, 1989.

[GL89]

R. I. Greenberg and C. E. Leiserson. Randomized Routing on Fat Trees. *Advances in Computing Research*, Volume 5, pages 345–374, 1989.

[HC83]

J. S. Huang and Y. C. Chow. Parallel Sorting and Data Partitioning by Sampling. In the *Proceedings of the 7th Computer Software and Applications Conference (COMPSAC '83)*, pages 627–631, November 1983.

[Her86]

D. Hergert. Microsoft Excel. *The Microsoft Desktop Dictionary and Cross-Reference Guide*, Microsoft Press, 1986.

[HKO+94]

S. Hinrichs, C. Kosak, D. R. O'Hallaron, T. M. Stricker, and R. Take. An Architecture for Optimal All-to-All Personalized Communication. In the *Proceedings of the 1994 Symposium on Parallel Algorithms and Architectures (SPAA '94)*, June 1994.

[HL92]

R. V. Hogg and J. Ledolter. *Applied Statistics for Engineers and Physical Scientists*, Second Edition. Macmillan Publishing Company, 1992.

[HPF]

D. Loveman, *High-Performance Fortran Proposal*. Presented at the Rice University High-Performance Fortran Forum, January 27, 1992. [Contact: `loveman@mpsg.enet.dec.com`, or ftp to `titan.cs.rice.edu`]

[Hsieh93]

W. Hsieh. *Automatic Methods for Optimizing Irregular Parallel Applications*. Ph.D. Proposal, November, 1993.

[HV81]

H. V. Henderson and P. F. Velleman. Building Multiple Regression Models Interactively. *Biometrics*. Volume 37, pages 391–411, 1981.

[HWW93]

W. C. Hsieh, P. Wang, and W. E. Weihl. *Computation Migration: Enhanced Locality for Distributed-Memory Systems*. In the *Proceedings of the 1993 Principles and Practice of Parallel Processing (PPoPP '93)*, May 1993.

[Intel91]

Intel Supercomputer Systems Division. *Paragon X/PS Product Overview*, March 1991.

[Jac88]
V. Jacobson. Congestion Avoidance and Control. In the *Proceedings of SIGCOMM '88*, August 1988.

[Jai91]
R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[Kus+94]
J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In the *Proceedings of 1994 International Symposium on Computer Architecture (ISCA '94)*, April 1994.

[KHM87]
M. Karol, M Hluchyj, and S. Morgan. Input versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communication*, Volume 35, Number 12, December 1987.

[Kle76]
L. Kleinrock. *Queuing Systems — Volume I: Theory*. John Wiley & Sons, 1976.

[KR88]
B. W. Kernigan and D. M. Ritchie. *The C Programming Language*, Second Edition. Prentice Hall, 1988.

[KTR93]
T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and Computation Performance of the CM-5. In the *Proceedings of Supercomputing '93*, pages 192–201, November 1993.

[Kun90]
H. T. Kung. An iWarp Multicomputer with an Embedded Switching Network. *Microprocessors and Microsystems*, Volume 14, Number 1, pages 59–60, January-February 1990.

[KWW+94]
F. M. Kaashoek, W. E. Weihl, D. A. Wallach, W. C. Hsieh, and K. L. Johnson. Optimistic Active Messages: Structuring Systems for High-Performance Communication. In the *Proceedings of the Sixth SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[LC94]
L. T. Liu and D. E. Culler. *Measurements of Active Messages Performance on the CM-5*. University of California Technical Report ???, date.

[Lei+92]
C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 1992 Symposium on Parallel and Distributed Algorithms*, June 1992.

[Lei85]
C. E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Computing. *IEEE Transactions on Computers*, Volume C-34, Number 10, pages 892–901, October 1985.

[Lin90]
M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 USENIX Summer Conference*, pages 211–220, June 1990.

[LLG+92]
D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, Volume 25, Number 3, pages 63–79, March 1992.

[LM86]
T. J. LeBlanc and J. M. Mellor-Crummey. *Debugging Parallel Programs with Instant Replay*. Technical Report TR194, University of Rochester, Computer Science Department, September 1986.

[LM88]
C. E. Leiserson and B. M. Maggs. Communication-Efficient Parallel Algorithms for Distributed Random-Access Machines. *Algorithmica*, Volume 3, pages 53–77, 1988.

[LM93]
K.-Q. Liao and L. Mason. Self-Optimizing Window Flow Control in High-Speed Data Networks. *Computer Communication*, Volume 16, Number 11, pages 706–716, November 1993.

[LMB92]
J. R. Levine, T. Mason and D. Brown. *lex & yacc*, Second Edition. O'Reilly & Associates, October 1992.

[Man:DJM]
A. Klietz. *Manual pages for the Distributed Job Manager*. Minnesota Supercomputer Center Incorporated, January 2, 1993.

[Man:grep]
Digital Equipment Corporation. grep(1), *Ultrix 4.3 General Information, Volume 3A, Commands(1): A-L*, 1993.

[Man:make]
Digital Equipment Corporation. make(1), *Ultrix 4.3 General Information, Volume 3B, Commands(1): M-Z*, 1993.

[MB76]
R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *IEEE Transactions on Computers*. Volume 19, Number 7, pages 395–404, July 1976.

[MTH90]
R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nel91]
G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[NM92]
R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In the *Proceedings of Supercomputing '92*, pages 502–511, August 1992.

[Pol+89]
C. D. Polychronopolous *et al.* Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In the *Proceedings of the 1989 International Conference on Parallel Processing*, Volume II, pages 39–48, August 1989.

[PBGB93]
G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. Integrated Building Blocks for Parallel Computers. In the *Proceedings of Supercomputing '93*, November 1993, pages 624–635.

[PM88]
S. K. Park and K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, Volume 31, Number 10, October 1988.

[PSW91]
C. Peterson, J. Sutton, and P. Wiley. iWarp — A 100-MOPS, LIW Microprocessor for Multicomputers. *IEEE Micro*, Volume 11, Number 3, June 1991.

[PTH90]
S. Pingali, D. Tipper, and J. Hammond. The Performance of Adaptive Window Flow Control's in a Dynamic Load Environment. In the *Proceedings of IEEE Infocom '90*, San Francisco, California, pages 55–62, June 1990.

[PTVF92]
W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, Second Edition. Cambridge University Press, 1992.

[RLW94]
S. K. Reinhart, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In the *Proceedings of the 21st International Symposium on Computer Architecture (ISCA '94)*, April 1994.

[RV87]
J. H. Reif and L. G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, Volume 34, Number 1, pages 60–76, January 1987. [Also *STOC '83*]

[Sch+86]
J. T. Schwartz *et al. Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Sam91]
A. Dain Samples. *Profile-Driven Compilation*. Ph.D. Dissertation, University of California at Berkeley, Technical Report UCB/CSD 91/627, April 1991.

[Sit92]
R. L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[SNN87]
J. H. Saltz, V. K. Naik, and D. M. Nicol. Reduction of the Effects on the Communication Delays in Scientific Algorithms on Message-Passing MIMD Architectures. *SIAM Journal of Scientific and Statistical Computing*, Volume 8, Number 1, pages 118–134, January 1987.

[Spe94]
P. Spector. *An Introduction to S and S-plus.* Duxbury Press, 1994.

[Ste90]
G. L. Steele, Jr. *Common Lisp, The Language,* Second Edition. Digital Equipment Corporation, 1990.

[Sus91]
A. Sussman. *Model-Driven Mapping onto Distributed-Memory Parallel Computers.* Ph.D. Dissertation, Carnegie-Mellon University, Technical Report CMU-CS-91-187, September 1991.

[Sus92]
A. Sussman. Model-Driven Mapping onto Distributed-Memory Parallel Computers. In the *Proceedings of Supercomputing '92*, pages 818–829, August 1992.

[Tan88]
A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1988.

[TLL93]
C. A. Thekkath, H. M. Levy and E. D. Lazowska. *Efficient Support for Multicomputing on ATM Networks.* Technical Report 93-04-03. Department of Computer Science and Engineering, University of Washington, April 12, 1993.

[TMC92]
Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary,* January 1992.

[TMC93]
Thinking Machines Corporation. *CMMD Reference Manual, Version 3.1,* October 1993.

[TPB+91]
K. R. Traub, G. M. Papadopoulos, M. J. Beckerle, J. E. Hicks, and J. Young. Overview of the Monsoon Project. In the *Proceedings of the 1991 IEEE International Conference on Computer Design (ICCD '92),* October 1991.

[vECGS92]
T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In the *Proceedings of the 1992 International Symposium on Computer Architecture (ISCA '92),* May 1992.

[WBC+91]
W. E. Weihl, E. A. Brewer, A. Colbrook, C. Dellarocas, W. C. Hsieh, A. Joseph, C. A. Waldspurger, and P. Wang. *Prelude: A System for Portable Parallel Software.* Technical Report MIT/LCS/TR-519, Massachusetts Institute of Technology, October 1991.

[Wol91]
S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*, Second Edition. Addison-Wesley Publishing Company, 1991.

[WR93]
Wolfram Research. *Guide to the Standard Mathematica Packages, Version 2.2.* Third Edition. Wolfram Research Technical Report, January 1993.

[WS91]
L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, 1991.

[Yel92]
K. Yelick. Programming Models for Irregular Applications. In the *Proceedings of the Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed-Memory Multiprocessors*, Boulder, Colorado, October 1992. Also *ACM SIGPLAN Notices*, Volume 28, Number 1, pages 17–20, January 1993.

Strata Reference Manual: Version 2.0A

Strata is a multi-layer communications library under development at MIT. Version 2.0 provides safe, high-performance access to the control network and flexible, high-performance access to the data network.

Strata works correctly on top of CMMD 3.0; both the data- and control-network operations can be mixed with Strata calls. However, Strata forms a complete high-performance communication system by itself, with substantial support for degugging and monitoring.

Strata provides several advantages over CMMD 3.0 [TMC93]:

Support for split-phase control-network operations. Split-phase operations allow the processor to perform work while the global operation completes. Strata provides split-phase barriers, segmented scans, and reductions.

Support for debugging and monitoring. Strata provides versions of `printf` that do not poll and can be used within handlers. This allows users to insert print statements for debugging without changing the atomicity of the surrounding code. Strata also provides routines for accurately timing short events, and for clean assertion checking and error handling. Finally, Strata provides support for automatic generation of state graphs, which depict the state of each processor versus time.

Support for developing high-performance data-communication protocols. Strata provides a rich and flexible set of data-network procedures that can be used to implement complex data-communication protocols. By understanding certain potential pitfalls, the Strata user can implement protocols that are more complex, are more robust, and achieve higher performance than those that can be implemented with the procedures and usage-rules of CMMD 3.0.

Higher performance. Table 1 compares the performance of Strata with that of CMMD 3.0. Strata's control-network procedures are considerably faster than CMMD's (though these operations rarely dominate execution time).

CMMD 3.0		Strata	
Primitive	**Time**	**Primitive**	**Time**
`CMMD_sync_with_nodes`	170 cycles	`Barrier`	140 cycles
`CMMD_scan_int`	380 cycles	`CombineInt`	150 cycles
`CMMD_scan_v`	241 cycles/word	`CombineVector`	49 cycles/word
`CMMD Broadcast, 1 word`	230 cycles	`Broadcast`	90 cycles
`CMMD Broadcast, double`	380 cycles	`BroadcastDouble`	90 cycles
`CMMD Broadcast, vector`	150 cycles/word	`BroadcastVector`	49 cycles/word
`CMAML_request`	67 cycles	`SendLeftPollBoth`	43 cycles
`CMAML_poll`	45 cycles	`PollBothTilEmpty`	22 cycles
`bzero`	36 cycles/word	`ClearMemory`	21 cycles/word

Table 1: Relative performance of Strata. The cycles counts for the active message procedures (`CMAML_request` and `SendLeftPollBoth`) as well as for the polling procedures (`CMAML_poll` and `PollBothTilEmpty`) are for the case when no messages arrive.

Contents

1 Using Strata

Strata programs have their own default host program and are linked with the Strata library. The Strata directory includes an example program (radix sort) that uses many of the Strata procedures.[1] Primary files:

File	Description
README	Installation notes
INSTALL	Installation script
`strata.h`	Defines all of the prototypes, typedefs and macros
`libstrata.a`	The Strata library, linked in with `-lstrata`
`libstrataD.a`	The Strata debugging library, linked in with `-lstrataD`
`strata_host.o`	The default host program object file
`Makefile`	The `make` file for the radix-sort application
`radix.c`	Radix-sort source code
`do-radix`	The DJM script for running radix sort
`control_net_inlines.c`	Inline control-net procedures, included by `strata.h`
`data_net_inlines.c`	Inline data-net procedures, included by `strata.h`
`strata_inlines.c`	Basic inline procedures, included by `strata.h`
`src`	Directory containing the Strata source code
`strata.ps`	The PostScript for this document.

See README for installation notes. After installation, users should be able to copy `Makefile`, `radix.c`, and `do-radix` into their own directory and build the example application. `Makefile` contains the necessary options to include `strata.h` and to link with the Strata library.

2 Strata Functionality

The rest of this document covers the current set of Strata routines. Figure 1 shows the structure of the Strata layering; each block corresponds to one of the following sections. Appendix A lists the prototypes for reference. There are several groups of Strata routines:

	Section	Description
3	Basics	Initialization, error handling, basic macros and globals
4	Control Network	Primitive control-network operations
5	Composite Reductions	Higher-level reductions
6	Data Network	Sending and receiving short data messages
7	Block Transfers	Sending and receiving blocks of data
8	Multi-Block Transfer	Higher-level data movement
9	Debugging	Support for debugging and timing
10	Graphics	Support for state graphs

[1]For MIT Scout users, the Strata library and include files are currently in `/u/brewer/strata`.

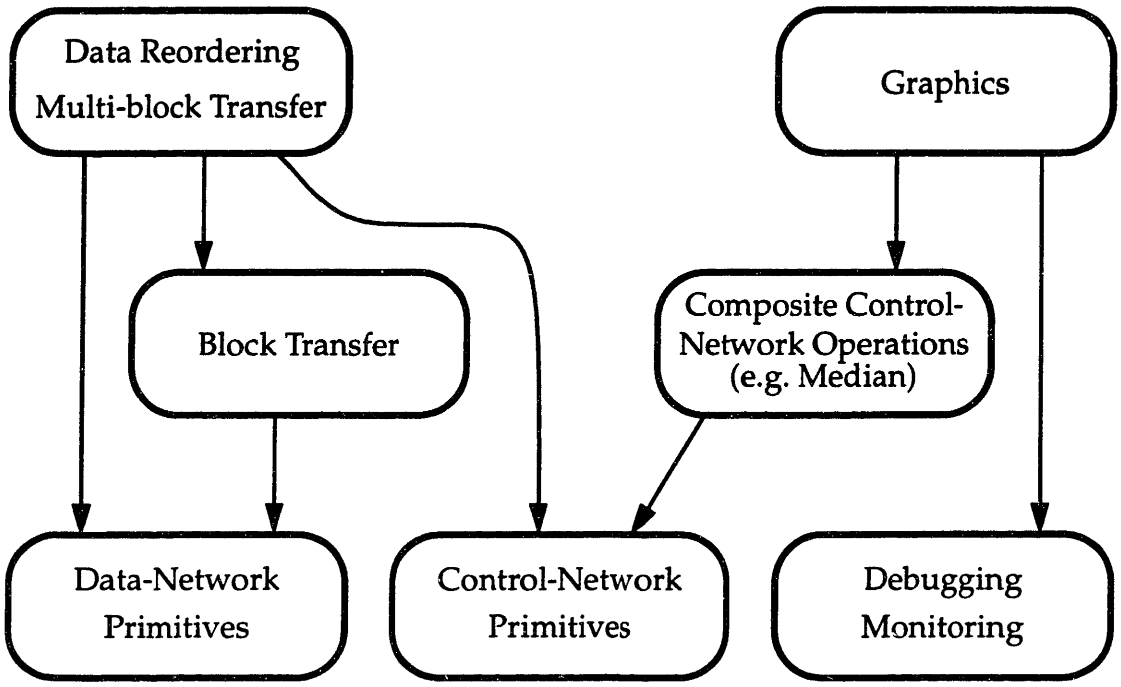


Figure 1: The layering of Strata.

3 Strata Basics

3.1 Type Qualifiers

ATOMIC	Used in function prototypes indicate that the procedure does not poll. This in only a convention; it has no effect on the procedure.
NORETURN	Used in function prototypes to indicate that the procedure never returns (like `exit`).
HANDLER	Used in function prototypes to indicate that the procedure is a handler. This in only a convention; it has no effect on the procedure.

3.2 Globals

`int Self`	Local node number
`int PartitionSize`	Current partition size
`int logPartitionSize`	The log (base two) of the partition size
`int PartitionMask`	Mask for legal node bits
`char *StrataVersion`	String: version number and date
`float StrataVersionNumber`	The version number

3.3 Types

`Word`	A generic 32-bit quantity (unsigned)
`DoubleWord`	A generic 64-bit quantity (unsigned)
`Bool`	Used for booleans; can be set to `True` (1) or `False` (0)

3.4 Macros

`LEFTMOST`	Defined as `(Self==0)`
`RIGHTMOST`	Defined as `(Self==PartitionSize-1)`

3.5 Procedures

`void StrataInit(void)`

This procedure initializes Strata. In particular, it:

1. Enables CMMD,
2. Disables interrupts,
3. Initializes the Strata global variables,
4. Sets `stdout` and `stderr` to independent append mode, and
5. Enables broadcasting.

`NORETURN StrataExit(int code)`

Exits the application after cleaning up. It is an error to exit without going through `StrataExit`, so Strata redefines `exit` and `assert` to exit through `StrataExit`. If `code` is non-zero, the entire application will exit immediately. If it is zero, then the node idles until all processors reach `StrataExit`, at which point the entire application exits normally. During the idle time, the node handles incoming messages (by polling).

`NORETURN StrataFail(const char *fmt, ...)`

The arguments work like `printf`. After printing the message to `stderr`, it exits through `StrataExit` with error code -1.

`ATOMIC void ClearMemory(void *region, unsigned length_in_bytes)`

This procedure zeroes out the region. It is functionally equivalent to `bzero`, but is faster because it uses double-word writes exclusively. It is provided primarily as an example of how to use double-word accesses from Gnu C. (Source code is in `memory.c`.)

3.6 Random-Number Generator

`ATOMIC unsigned Random(void)`

Returns a pseudo-random 32-bit value. This is the "minimal standard" random-number generator described by Park and Miller [PM88]. The default seed is `Self+1` (set during `StrataInit`), which gives each processor a unique but repeatable sequence.

ATOMIC unsigned SetRandomSeed(unsigned seed)

Sets the random seed for this node to `seed`. If `seed` is the special value `TIME_BASED_SEED`, then the seed is set based on the cycle counter and the processor number. This results in different seeds for different runs (as well as for different nodes). The return value is the seed actually used, which can be used later to repeat the sequence.

4 Control-Network Primitives

4.1 Barriers

void Barrier(void)

Returns when all nodes reach the barrier. It is equivalent to `GlobalOR(False)`.

ATOMIC void StartBarrier(void)

Starts a split-phase barrier.

ATOMIC Bool QueryBarrier(void)

Returns true if and only if the last barrier has completed.

void CompleteBarrier(void)

Returns when the last barrier has completed, possibly blocking.

4.2 Global OR

Bool GlobalOR(Bool not_done)

Returns the OR of the `not_done` argument of each PE. All nodes must participate.

ATOMIC void StartGlobalOR(Bool not_done)

Starts a global OR operations, but returns immediately.

ATOMIC Bool QueryGlobalOR(void)

Returns true if and only if the last global OR has completed.

Bool CompleteGlobalOR(void)

Returns the result of the last global OR; blocks until the result is available.

ATOMIC void SetAsyncGlobalOR(Bool not_done)

This is an asynchronous version of the global-OR operation. This procedure sets this processor's contribution. During `StrataInit`, each node's value is set to `True`.

ATOMIC Bool GetAsyncGlobalOR(void)

Reads the asynchronous global OR bit. Returns true if and only if at least one node has its asynchronous-global-OR bit set to true.

4.3 Combine Operations

The control network provides five associative functions for combine operations: signed add, or, xor, unsigned add, and signed max. Thus, there are fifteen primitive combine operations:

ScanAdd	BackScanAdd	ReduceAdd
ScanOr	BackScanOr	ReduceOr
ScanXor	BackScanXor	ReduceXor
ScanUadd	BackScanUadd	ReduceUadd
ScanMax	BackScanMax	ReduceMax

The `Scan` variants perform a forward scan (by processor number), the `BackScan` variants perform a backward scan, and the `Reduce` variants perform a reduction. Together these fifteen variants form the enumerated type `CombineOp`, which is used by all of the Strata combine primitives.

4.3.1 Segmented Scans

Strata also offers segmented scans. The segments are global state maintained by the hardware; Strata scans implicitly use the current segments. Initially, the partition is set up as one large segment covering all nodes. Note that reduction operations ignore the segments (this is a property of the hardware), but segmented reductions can be built out of two segmented scans. Normally, a segmented scan involves calling `SetSegment` followed by one of the combine functions, but the call to `SetSegment` is often left out, since segments are typically used more than once. Finally, segments are determined at the time the scan starts (on this node); changing the segments after that has no effect.

There are three kinds of segment boundaries; they form `SegmentType`:

```
typedef enum {
    NoBoundary, ElementBoundary, ArrayBoundary
} SegmentType;
```

The difference between `ElementBoundary` and `ArrayBoundary` requires an example. Assume that there are 16 processors (0 on the left), the operation is `ScanAdd`, and that segment boundaries are marked as N, E, or A:

```
Case 1:
    Value:     1 1 1 1   2 2 2 2   3 3 3 3   4 4 4 4
    Segment:   E N N N   E N N N   E N N N   E N N N

    Output:    0 1 2 3   0 2 4 6   0 3 6 9   0 4 8 12

Case 2:
    Value:     1 1 1 1   2 2 2 2   3 3 3 3   4 4 4 4
    Segment:   A N N N   A N N N   A N N N   A N N N

    Output:    0 1 2 3   4 2 4 6   8 3 6 9  12 4 8 12
```

Case 1 outputs the expected answer; the first node in a segment receives the identity value. However, case 2 is also useful (and matches the hardware). In case 2, the first node of a segment receives the value of the previous segment, which in this example is the sum of the previous segment. Thus, `ArrayBoundary` essentially means "give me the value from the previous processor, but don't forward that value to the next processor". This semantics is required when performing a scan on a "vector" that is wider than the partition and thus uses multiple elements per node. For example, a scan of a 64-element wide vector on 16 processors uses 4 elements per node. If the intended boundary is between the 2^{nd} and 3^{rd} elements, then the first two elements depend on the value from the previous node, and only the third element gets the identity value. The test code, `test-all.c`, uses both forms of segments.

`ATOMIC void SetSegment(SegmentType boundary)`

Sets the segment status for this node to `boundary`. If all nodes execute `SetSegment(NoBoundary)`, then the nodes form one large segment; this is the initial segment setting. It is fine for only a subset of the nodes to call `SetSegment`. Note that reductions (both regular and composite) ignore segment boundaries.

`ATOMIC SegmentType CurrentSegment(void)`

Returns the current segment setting for this node.

4.3.2 Combine Procedures

`int CombineInt(int value, CombineOp kind)`

Performs a global combine operation using variant `kind`; each PE contributes `value`. The return value is the result of the combine operation. This implicitly uses the current segment settings.

`ATOMIC void StartCombineInt(int value, CombineOp kind)`

Starts a split-phase combine operation. This implicitly uses the current segment settings.

`ATOMIC Bool QueryCombineInt(void)`

Returns true if and only if the last combine operation has completed.

`int CompleteCombineInt(void)`

Returns the result of the pending split-phase combine operation, possibly blocking.

```
void CombineVector(int to[], int from[],
    CombineOp kind, int num_elements)
```

Performs a vector combine operation. The input vector is `from` and the result is placed into `to`, which may be the same as `from`. This implicitly uses the current segment settings. (The order of `from` and `to` matches that of `memcpy` and its variants.)

```
void StartCombineVector(int to[], int from[],
     CombineOp kind, int num_elements)
```

Starts a split-phase vector combine operation. This implicitly uses the current segment settings.

`void CompleteCombineVector(void)`

Blocks until the pending split-phase vector combine operation completes. There is currently no `QueryCombin-eVector`.

4.4 Broadcast Operations

`ATOMIC void Broadcast(Word value)`

Broadcasts one word to all nodes.

`ATOMIC Bool QueryBroadcast(void)`

Returns true if and only if the pending broadcast has completed. This works with `BroadcastDouble` as well.

`Word ReceiveBroadcast(void)`

Returns the broadcast word. Should be called on all nodes except for the one that performed the broadcast.

`ATOMIC void BroadcastDouble(double value)`

Broadcast a double to all nodes.

`double ReceiveBroadcastDouble(void)`

Return the broadcast double. Should be called on all nodes except for the one that performed the broadcast.

`void BroadcastVector(Word to[], Word from[], int num_elements)`

Broadcast a vector of words. Fills in `to` as the words are received. Returns when all words are sent; not all may have been received.

`void ReceiveBroadcastVector(Word to[], int num_elements)`

Receive a vector of words into array `to`. Should be called on all nodes except for the one that performed the broadcast. For the broadcaster, the `to` array should be same as the call to `BroadcastVector`. Returns when all of the words have been received.

5 Composite Reductions

There are many useful operations that can be built on top of the CM-5's control-network hardware. Currently, Strata provides five such operations for three basic types: integers, unsigned integers, and single-precision floating-point numbers. These operations form the enumerated type `CompositeOp`:

```
typedef enum {
    Min, Max, Average, Variance, Median
} CompositeOp;
```

As with CMMD and the Strata control-network primitives, these reductions ignore the segment boundaries: all nodes form one large segment.[2] `Min`, `Max`, and `Average` are straightforward reductions. `Median` computes the median of the values contributed by the nodes. If there are an even number of values, then it returns the smaller of the two middle values.

[2]Future versions of Strata may provide segmented reductions; let us know if it is important to you.

`Variance` computes the sample variance of the values:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}) = \frac{1}{n-1}\left[\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}\right]$$

where n is the size of the partition, $\bar{x}$ is the average, and x_i is the value contributed by the i^{th} node. The latter form is the one actually used. If the partition size is one, the reduction returns zero. The sample standard deviation is square root of the returned value.

There are three composite reduction procedures, one for each basic type:

`int CompositeInt(int value, CompositeOp op)`

This procedure applies the composite operation to integer values. The `Average` operation has the same overflow properties as a `ReduceAdd` of the same values. `Variance` is limited to the same precision as `float`.

`unsigned CompositeUint(unsigned value, CompositeOp op)`

This procedure applies the composite operation to unsigned integer values. The `Average` operation has the same overflow properties as a `ReduceUadd` of the same values. `Variance` is limited to the same precision as `float`.

`float CompositeFloat(float value, CompositeOp op)`

This procedure applies the composite operation to single-precision floating point values. It uses the CMMD procedure `CMMD_reduce_float` to ensure IEEE floating-point compatibility, and is provided primarily for completeness.

6 Data-Network Primitives

On the CM-5, an active message is a single-packet message in which the first word of the message specifies a procedure on the receiving processor. This procedure is called a *handler* and the handler's job is to remove the message from the network and incorporate it into the ongoing computation[vCGS92]. A CM-5 active-message packet consists of five words.

handler	arg1	arg2	arg3	arg4

When the receiving node polls the network and discovers an active message, the polling procedure invokes the `handler` with the four arguments `arg1`, `arg2`, `arg3`, and `arg4` as parameters.

6.1 Sending Active Messages

The CM-5 data network actually consists of two separate networks: the *right* network and the *left* network. Strata provides procedures to send and receive active messages on either network.

`void SendBothRLPollBoth(int proc, void (*handler)(), ...)`

Send an active message to processor `proc` using either network, and polling both networks. The active message is formed with `handler` as the first word and the remaining (up to) four parameters as `arg1`, `arg2`, `arg3`, and `arg4`. This procedure starts by trying to send the message on the right network and polling the right network. If the message does not get out, then it tries to send on the left network and polls the left network. This procedure continues this cycle until the message gets sent.

`void SendBothLRPollBoth(int proc, void (*handler)(), ...)`

This procedure starts by trying to send its message on the left network and polling the left network. If unsuccessful, it tries to send on the right network and polls the right network. This procedure repeats this cycle until the message gets sent.

`void SendBothRLPollRight(int proc, void (*handler)(), ...)`

This procedure sends an active message using either network, but it only polls the right network. It starts by trying to send its message on the right network and polling the right network. If the message doesn't get sent, then it tries to send on the left network (but doesn't poll the left network). This procedure repeats this cycle until the message gets sent.

`void SendBothLRPollRight(int proc, void (*handler)(), ...)`

This procedure starts by trying to send its message on the left network (but it doesn't poll the left network). If unsuccessful, it tries to send on the right network and polls the right network. This procedure repeats this cycle until the message gets sent.

`void SendLeftPollBoth(int proc, void (*handler)(), ...)`

This procedure sends an active message on the left network and polls both networks. It starts by trying to send its message on the left network and polling the left network. If the message does not get sent, then it polls the right network. This procedure repeats this cycle until the message gets sent.

Roughly, these procedures have the following CMMD 3.0 equivalents.

CMMD 3.0	Strata
`CMAML_request`	`SendLeftPollBoth`
`CMAML_reply`	`SendBothRLPollRight`
`CMAML_rpc`	`SendBothRLPollBoth`

(Actually `CMAML_reply` will only send its message on the right network.) When using any of the `SendBoth` variants, the choice between `SendBothRL` and `SendBothLR` can be made arbitrarily, but if these procedures are being called many times (especially in a tight loop), then higher performance is achieved by alternating between them.

6.2 Receiving messages

Messages are removed from the network and appropriate handlers are invoked by the following polling procedures.

`void PollLeft(void)`

This procedure checks the left network, and if there is a pending message, it removes the message and calls the appropriate handler.

`void PollRight(void)`

This procedure checks the right network, and if there is a pending message, it removes the message and calls the appropriate handler.

`void PollBoth(void)`

This procedure checks both networks and pulls out at most one message from each. It is equivalent to `PollLeft` followed by `PollRight`.

`void PollLeftTilEmpty(void)`

This procedure keeps removing messages from the left network and invoking the appr^priate handlers until it finds that no further messages are pending.

`void PollRightTilEmpty(void)`

This procedure keeps removing messages from the right network and invoking the appropriate handlers until it finds that no further messages are pending.

`void PollBothTilEmpty(void)`

This procedure keeps removing messages from both networks and invoking the appropriate handlers until it finds that no further messages are pending on either network.

`void PollLeftThenBothTilEmpty(void)`

This procedure begins by polling the left network. If it finds no message pending, then it returns. If it finds a pending message, then it continues polling both networks until it finds no further messages are pending on either network.

`void PollRightThenBothTilEmpty(void)`

This procedure begins by polling the right network. If it finds no message pending, then it returns. If it finds a pending message, then it continues polling both networks until it finds no further messages are pending on either network.

In general, frequent polling with `PollBothTilEmpty` is recommended. In a loop, however, higher-performance is achieved by alternating between `PollLeftThenBothTilEmpty` and `PollRightThen-BothTilEmpty`.

7 Block Transfers

Block transfers are performed by sending special messages called *Xfer messages* special data structures called *ports* at the receiving processor. A port is a structure defined in Strata as follows.

```
typedef struct {
    Word  *base;
    int    count;
    void (*handler)();
    int    user1;
    int    user2;
    int    user3;
    int    user4;
    int    user5;
} StrataPort;
```

Strata provides each processor with a global array of STRATA_NUM_PORTS ports (currently 4096). This array is called the `StrataPortTable` and is defined as

```
StrataPort StrataPortTable[STRATA_NUM_PORTS];
```

Ports are designated by number, that is, by index into the `StrataPortTable`. To receive a block transfer at a port, the port's `base` field must be initialized to point to a block of memory into which the block transfer data can be stored. Also, the port's `count` field must be set to the number of words to be received in the block transfer. As the block transfer data arrives, the count value gets decremented, and when the count reaches zero, the procedure specified by the `handler` field (if not NULL) gets invoked with the port number as its single parameter. The other five fields in the port structure, `user1` through `user5`, are available for arbitrary use.

Block transfer data is received by the same polling procedures that receive active messages as described in the previous section.

The following two procedures are used to send a block transfer.

`void SendBlockXferPollBoth(int proc, unsigned port_num, int offset, Word *buffer, int size)`

Sends `size` words starting at `buffer` to the port `port_num` at the destination processor `proc` and stores the data in the destination processor's memory starting at the address given by adding `offset` (in words) to the port's base value, that is, at address (`StrataPortTable[port_num].base + offset`). The data is sent on both networks, and this procedure will poll both networks.

`void SendBlockXferPollRight(int proc, unsigned port_num, int offset, Word *buffer, int size)`

Sends `size` words starting at `buffer` to the port `port_num` at the destination processor `proc` and stores the data in the destination processor's memory starting at the address given by adding `offset` (in words) to the port's base value, that is, at address (`StrataPortTable[port_num].base + offset`). The data is sent on both networks, but this procedure only polls the right network.

The port number, `port_num`, must be between 0 and `STRATA_NUM_PORTS - 1`. The `offset` is a signed value and must be at least STRATA_MIN_XFER_OFFSET (currently -2^{19}), and the sum `offset + size` must be no larger than STRATA_MAX_XFER_OFFSET (currently $2^{19} - 1$). Thus, a block transfer can consist of up to 2^{20} words (with the port's `base` field pointing to the middle of the destination block). (To send a larger block, use more than one port.)

Notice that all quantities are in terms of words not bytes.

The block transfer is most efficient when the source block address (given by `buffer`) and the destination block address (given by adding `offset` to the port's `base` value) have the same alignment. That is, when both are `DoubleWord` aligned or both are not `DoubleWord` aligned (but are, of course, `Word` aligned).

Before sending a block transfer, the receiver must have set the `base` field of the target port, but the `count` and `handler` fields can be set by the sending processor before, after, or during the block transfer. The sending processor sets these fields by sending an ordinary active message (as described in the previous section) that invokes a handler on the target processor to set these fields. Strata provides just such a handler.

`HANDLER void StrataXferHeaderHandler(int port_num, int size, void (*handler)())`

This handler increments the `count` field of port `port_num` by `size`, and if the port's `handler` field is NULL, sets the `handler` field to `handler`. It then checks to see if the `count` field is zero, and if so, it invokes the procedure given by the `handler` field (if not NULL) with `port_num` as its single parameter.

If the sending processor is going to set the port's `handler` field, then the destination processor should initialize the port's `handler` field to NULL. And if the sending processor is going to set the port's `count` field, then the destination processor should initialize the port's `count` field with zero. The sending processor then, in addition to calling `SendBlockXferPollBoth` or `SendBlockXferPollRight` to send the actual data block, must send an active message that invokes `StrataXferHeaderHandler`. Some of the block data may arrive at the destination processor before this active message, and in this case, the `count` field will actually go negative (it is initialized at zero). Then when the active message arrives, the `count` field gets incremented — hopefully to some non-negative value. Of course, the active message may arrive even after *all* of the block transfer data; in this case, incrementing the `count` field should bring it to zero, and for this reason, `StrataXferHeaderHandler` checks to see if the `count` field is zero and takes appropriate action.

As a typical example, a receiving processor might set the `base` field of a port to point to a (suitably large) buffer, `dest_buffer`, and initialize the port's `count` field to 0 and its `handler` field to NULL. Then the sending processor can send `size` words from its `source_buffer` to the receiving processor's `dest_buffer` (assuming the sending processor knows that the port is number `port_num`) and cause the procedure `handler` to be invoked on the receiver when the transfer is complete with the following pair of procedures.

```
SendBothRLPollBoth(dest_proc, StrataXferHeaderHandler, port_num,
                   size, handler);
SendBlockXferPollBoth(dest_proc, port_num, 0, source_buffer, size);
```

7.1 For the curious

The procedures `SendBlockXferPollBoth` and `SendBlockXferPollRight` send their data through a sequence of single-packet Xfer messages. An Xfer message manages to pack 4 words of payload data into the 5 word CM-5 data-network packet.

port_and_offset	data1	data2	data3	data4

The port number and offset values are packed into a single word `port_and_offset`. When such an Xfer message arrives at its destination processor (and the destination processor receives it by polling), a special Xfer handler routine is invoked. This routine splits the `port_and_offset` into its two components: `port_num` and `offset`. Then it stores the four data words, `data1` through `data4`, at consecutive addresses starting at the address given by adding `offset` (in words) to the port's base address, that is, at address `StrataPortTable[port_num].base + offset`. It then subtracts 4 from the port's `count` field, and takes appropriate action if the `count` is zero.

Port number and offset values are packed into and unpacked from a single word with the following functions.

`ATOMIC unsigned PortAndOffsetCons(unsigned port_num, int offset)`

Packs `port_num` and `offset` into a single word. The `port_num` must be between 0 and `STRATA_NUM_PORTS - 1`, and the `offset` must be between STRATA_MIN_XFER_OFFSET and STRATA_MAX_XFER_OFFSET. (Currently, `port_num` lives in the low 12 bits and `offset` lives in the high 20 bits of `port_and_offset`, so STRATA_NUM_PORTS equals 4096, STRATA_MIN_XFER_OFFSET equals -2^{19}, and STRATA_MAX_XFER_OFFSET equals $2^{19}-1$.

`ATOMIC unsigned PortAndOffsetPort(unsigned port_and_offset)`

Extracts the `port_num` from `port_and_offset`.

`ATOMIC int PortAndOffsetOffset(unsigned port_and_offset)`

Extracts the `offset` from `port_and_offset`.

Single-packet Xfer messages are sent with the following procedures.

`void SendXferBothRLPollBoth(int proc, unsigned port_and_offset, Word data1, Word data2, Word data3, Word data4)`

```
void SendXferBothLRPollBoth(int proc, unsigned port_and_offset,
     Word data1, Word data2, Word data3, Word data4)
```

```
void SendXferBothRLPollRight(int proc, unsigned port_and_offset,
     Word data1, Word data2, Word data3, Word data4)
```

```
void SendXferBothLRPollRight(int proc, unsigned port_and_offset,
     Word data1, Word data2, Word data3, Word data4)
```

Each single-packet Xfer message carries exactly four words of payload. For a block transfer with a number of words that is not a multiple of four, extra messages must be sent. These extra messages can be sent with ordinary active messages since they only need 1, 2, or 3 words of payload. Strata provides handlers to deal with these active messages.

```
HANDLER StrataXferPut3Handler(unsigned port_and_offset,
     Word data1, Word data2, Word data3)
```

```
HANDLER StrataXferPut2Handler(unsigned port_and_offset,
     Word data1, Word data2)
```

```
HANDLER StrataXferPut1Handler(unsigned port_and_offset,
     Word data1)
```

These handlers store the data words into the appropriate location, subtract the appropriate value from the port's `count` field, and take appropriate action if the `count` is zero.

None of these functions, procedures, or handlers are needed if you use `SendBlockXferPollBoth` or `SendBlockXferPollRight`. These two procedures do it all for you.

8 Multi-Block Transfer

Strata provides an asynchronous block-transfer protocol to support multiple block transfers. This interface allows Strata to interleave packets from several different block transfers, which improves the efficiency of the network and increases the net effective bandwidth.

Pending transfers are identified with handles of type `ABXid`.

```
ABXid AsyncBlockXfer(int proc, unsigned port,
     int offset, Word *buffer, int size,
     void (*complete)(ABXid id, Word *buffer))
```

Initiate an asynchronous block transfer to port `port` of processor `proc`. The argument correspond to normal block-transfer routines, except for the additional argument `complete`. This function, if non-NULL, is called upon completion of the *sending* of this transfer. The return value is an identifier that is used by the following routines.

`void ServiceAllTransfers(int rounds)`

This procedure services all of the pending asynchronous block transfers, sending `2*rounds` packets for each transfer. If `rounds` is -1, then this procedures synchronously completes all pending transfers. The preferred use of this routine is to set up all of the transfers (or a reasonable size subset), and then to call `ServiceAllTransfers(-1)` to actually send them. For target distributions that prevent explicit scheduling of transfers, this technique provides about doubles the performance of just sending the transfers synchronously.

`void CompleteTransfer(ABXid id)`

This synchronously completes the named transfer.

```
int GetABXproc(ABXid id)
unsigned GetABXport(ABXid id)
int GetABXoffset(ABXid id)
Word *GetABXbuffer(ABXid id)
int GetABXremaining(ABXid id)
```

These routines access the corresponding information regarding the named transfer.

9 Debugging Operations

9.1 Printing in Handlers

`ATOMIC int Qprintf(const char *format, ...)`

Queue up a printf to `stdout`. This can be used anywhere, including within handlers, because it does not poll and does not allocate memory. The return value is the number of characters in the message. `Qprintf` allows you to insert debugging statements *without* changing the atomicity of the surrounding code.

`ATOMIC int Qfprintf(FILE *out, const char *format, ...)`

Queue up an fprintf to file `out`. Otherwise identical to `Qprintf`.

`int EmptyPrintQ(void)`

Outputs the queued messages. This does poll and thus should not be called from within a handler. The return value is the number of messages that had been queued up.

A Note On pndbx

When using pndbx to debug a Strata application, it is often useful to read the queued-up print calls. Strata provides two global variables to help with this. First, the variable `char *StrataQueue` points to the buffer containing the queued-up text. The strings are null-terminated and packed continuously. Executing print StrataQueue from pndbx prints the first string (only). The variable `int StrataQueueLength` contains the number of queued-up strings. To see all of the strings, first use print &StrataQueue[0] to get an address, say 0x1cb4a0, then use 0x1cb4a0/256c to dump the buffer as list of 256 characters (the 256 is arbitrary — large queues may require a larger number). Finally, if the program is active, it is often safe to execute call EmptyPrintQ() to output the strings; this is less useful if the output is going to a file rather than the terminal.

9.2 Assertion Checking

Macro **assert(x)**

Calls `StrataFail` with a nice error message if `x` evaluates to 0. This replaces the normal `assert` macro both to improve the message and to ensure a clean exit.

9.3 Debugging Mode and Logging

Strata provides a debugging mode that performs safety checks and atomic logging. To use debugging mode, you must compile with the `-DSTRATA_DEBUG` flag and link with `-lstrataD` instead of `-lstrata`. Debugging mode uses several global variables:

`Bool StrataLogging`	True iff linked with the debugging library.
`Bool LogBarrier`	Log barrier info, default is `False`
`Bool LogBroadcast`	Log broadcast info, default is `False`
`Bool LogCombine`	Log combine info, default is `False`
`Bool LogQprintf`	Log Qprintf info, default is `False`

The log appears in file `CMTSD_printf.pn.`*number*, where *number* is the process id of the host. Strata programs print out the exact name on exit (unless killed). The log is always up to date; for example, when a program hangs, the last log entry for each node often reveals the nature of the problem. Strata comes with a perl script called `loglast` that outputs the last entry for each node.

Users may log their own messages in addition to those provided by the debugging library:

ATOMIC void StrataLog(const char *fmt, ...)

The printf-style message is added to the log. Note that all log entries are prepended with the processor number, so there is no need explicitly print it. `StrataLog` may be used anywhere, even when not using the debugging library. It should be viewed as an expensive function.

9.4 Timing Functions

ATOMIC INLINE unsigned CycleCount(void)

Returns the value of the cycle counter.

ATOMIC INLINE unsigned ElapsedCycles(unsigned start_count)

Returns the elapsed time in cycles given the starting time. The intended use is:

```
start = CycleCount();
...
elapsed = ElapsedCycles(start);
```

The routines are calibrated so that if used in an application compiled with `-O`, the overhead of the two calls is exactly subtracted out. Thus if the two calls are adjacent, the elapsed time is zero.[3] These routines will break if the code is time sliced between the calls, but this is extremely unlikely for (the intended) short timings. For example, a 10,000 cycle event has roughly a 1 in 330 chance of being time sliced given the usual time-slice interval of 0.1 seconds (and zero chance if the machine is in dedicated mode).

`ATOMIC INLINE double CyclesToSeconds(unsigned cycles)`

Converts a time in cycles into seconds (not microseconds).

`DoubleWord CurrentCycle64(void)`

Returns the number of cycles elapsed since initialization (via `StrataInit`). This is not as accurate as `ElapsedCycles`, but it works across time slices (modulo some OS bugs) and provides 64-bit resolution. The implementation uses the same underlying code as the CMMD timers and is thus exactly as accurate.

`ATOMIC INLINE unsigned CurrentCycle(void)`

Same as `CurrentCycle64` except that it returns the elapsed cycles since the initialization or the last call to `ResetCurrentCycle`. Like `ElapsedCycles` this procedure includes cycles due to time slicing; it is thus primarily useful in dedicated mode.

`void ResetCurrentCycle(void)`

This sets the `CurrentCycle` base time to zero. It is a synchronous operation and must be called on all nodes. The primary use of this routine is to get 32 bits of useful times in the middle of a program that runs for more the 2^32 cycles. It affects the timestamps used by the graphics module, which allows users to get a 2^32-cycle state-graph of their long-running program for any single such continuous interval.

10 Graphics

Version 2.0 supports a subset of the PROTEUS graphics capabilities. The primary form of graph currently supported by Strata is the *state graph*, which plots the state of each processor (as a color) versus time (in cycles). There are several limitations on state graphs: 1) there can be at most 16 states, 2) the program can run for at most 2^32 cycles after the call to `InitTrace`, and 3) there is a limit on the number of state changes per processor (although there are workarounds for this one).

The graphics module creates a trace file that is read by the program `stats`. The interpretation of the file is determined by a *graph-file specification* that specifies the names of states and how to build each graph (normally there is only one graph for Strata).

The easiest way to understand Strata's graphics is to play with the example radix-sort program (`radix.c`). It generates a trace file called `radix.sim` that can be examined with:

[3]With `-O`, the null timing case compiles to two consecutive reads of the cycle counter; the subtraction assumes this case and the result is almost always zero. The exception is when the two reads are in different cache lines and the cache line of the second read is not loaded. In this case, the timing includes the cache miss and typically returns 28 or 29 cycles. The point of all this is that you should be aware of cache behavior for short timings.

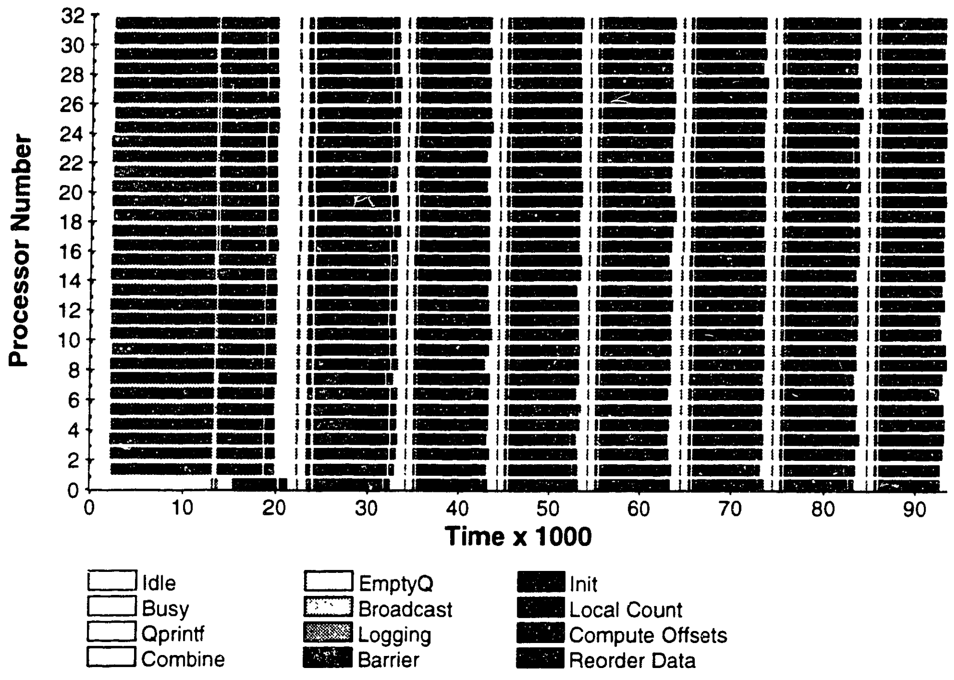


Figure 2: The state graph for radix sort.

```
stats -f radix.sim -spec cmgraph
```

The second pair of arguments identifies the graph-specification file. Figure 2 shows the state graph for radix sort. A seperate document covers the use of `stats` and the graph-specification language; `stats -help` provides some information as well.

State 0 always means "idle" and state 1 always means "busy", although the real meaning of these states is up to the application. In debugging mode, Strata uses states 2 through 7 as follows:

State	Meaning
2	Logging
3	Qprintf
4	EmptyPrintQ
5	Broadcasting
6	Combine Operation
7	Barrier

These states can be redefined via `strata.h` and may even be combined if the user needs more application-specific states.

Generating the trace file involves only three procedures:

`void InitTrace(const char *file, const char *title)`

This initializes the graphics module; it must be called before `StrataInit` and it must be called on all processors. It is a global operation. The first argument is the name of trace file. The second argument is the title of the simulation; if non-NULL, this title will appear as a subtitle on the graphs (the primary title is determined by the graph-specification file).

`ATOMIC int SetState(int state)`

This changes the state of this processor to `state` and returns the old state. The state change is timestamped with the current cycle time.

`Bool RecordStates(Bool on)`

Turns state recording (via `SetState`) on or off (for the local node only). This is primarily useful for turning off state generation after the interesting part of the program completes. For example, to record states for one section in the middle of a long-running program, use `ResetCurrentCycle()` to mark the beginning of the section, and `RecordStates(False)` to mark the end.

`void OutputStateLog(void)`

Normally, the state changes are kept in local memory until the program exits, at which point they are moved to disk. Currently, the buffering allows at most 2048 state changes. `OutputStateLog` empties the buffer so that more events can be generated. This is a synchronous operation and must be executed on all nodes; it should be viewed as a very expensive barrier. Most programs do not need to call this procedure.

11 Acknowledgments

Thanks to Mike Halbherr, Chris Joerg, Ulana Legedza, Tom Leighton, Charles Leiserson, Arthur Lent, Frans Kaashoek, Bradley Kuszmaul, Dave Park, Ryan Rifkin, Carl Waldspurger, Bill Weihl, and Yuli Zhou.

References

[PM88] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10), October 1988.

[TMC93] Thinking Machines Corporation. *CMMD Reference Manual, Version 3.0*, May 1993.

[vCGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the* 19^{th} *International Symposium on Computer Architecture (ISCA '92)*, pages 256–266, May 1992.

A Prototype Summary

Page	Basics
178	`void StrataInit(void)`
178	`NORETURN StrataExit(int code)`
178	`NORETURN StrataFail(const char *fmt, ...)`
178	`ATOMIC void ClearMemory(void *region, unsigned length_in_bytes)`
178	`ATOMIC unsigned Random(void)`
179	`ATOMIC unsigned SetRandomSeed(unsigned seed)`

	Timing
190	`ATOMIC unsigned CycleCount(void)`
190	`ATOMIC unsigned ElapsedCycles(unsigned start_count)`
191	`ATOMIC double CyclesToSeconds(unsigned cycles)`
191	`DoubleWord CurrentCycle64(void)`
191	`ATOMIC unsigned CurrentCycle(void)`
191	`void ResetCurrentCycle(void)`

	Global OR
179	`Bool GlobalOR(Bool not_done)`
179	`ATOMIC void StartGlobalOR(Bool not_done)`
179	`ATOMIC Bool QueryGlobalOR(void)`
179	`Bool CompleteGlobalOR(void)`
179	`ATOMIC void SetAsyncGlobalOR(Bool not_done)`
180	`ATOMIC Bool GetAsyncGlobalOR(void)`

	Barriers
179	`void Barrier(void)`
179	`ATOMIC void StartBarrier(void)`
179	`ATOMIC Bool QueryBarrier(void)`
179	`void CompleteBarrier(void)`

	Combine Operations
181	`ATOMIC void SetSegment(SegmentType boundary)`
181	`ATOMIC SegmentType CurrentSegment(void)`
181	`int CombineInt(int value, CombineOp kind)`
181	`ATOMIC void StartCombineInt(int value, CombineOp kind)`
181	`ATOMIC Bool QueryCombineInt(void)`
181	`int CompleteCombineInt(void)`
181	`void CombineVector(int to[], int from[], CombineOp kind, int num_elements)`
181	`void StartCombineVector(int to[], int from[], CombineOp kind, int num_elements)`
181	`void CompleteCombineVector(void)`
183	`int CompositeInt(int value, CompositeOp op)`
183	`unsigned CompositeUint(unsigned value, CompositeOp op)`
183	`float CompositeFloat(float value, CompositeOp op)`

	Broadcast Operations
182	`ATOMIC void Broadcast(Word value)`

```
182   ATOMIC Bool QueryBroadcast(void)
182   Word ReceiveBroadcast(void)
182   ATOMIC void BroadcastDouble(double value)
182   double ReceiveBroadcastDouble(void)

182   void BroadcastVector(Word to[], Word from[], int num_elements)
182   void ReceiveBroadcastVector(Word to[], int num_elements)
```

Sending Active Messages

```
183   void SendBothRLPollBoth(int proc, void (*handler)(), ...)
184   void SendBothLRPollBoth(int proc, void (*handler)(), ...)
184   void SendBothRLPollRight(int proc, void (*handler)(), ...)
184   void SendBothLRPollRight(int proc, void (*handler)(), ...)
184   void SendLeftPollBoth(int proc, void (*handler)(), ...)
```

Polling

```
184   void PollLeft(void)
184   void PollRight(void)
184   void PollBoth(void)
185   void PollLeftTilEmpty(void)
185   void PollRightTilEmpty(void)
185   void PollBothTilEmpty(void)
185   void PollLeftThenBothTilEmpty(void)
185   void PollRightThenBothTilEmpty(void)
```

Block Transfers

```
186   void SendBlockXferPollBoth(int proc, unsigned port_num,
                 int offset, Word *buffer, int size)
186   void SendBlockXferPollRight(int proc, unsigned port_num,
                 int offset, Word *buffer, int size)

187   ATOMIC unsigned PortAndOffsetCons(unsigned port, int offset)
187   ATOMIC unsigned PortAndOffsetPort(unsigned port_and_offset)
187   ATOMIC int PortAndOffsetOffset(unsigned port_and_offset)

187   void SendXferBothRLPollBoth(int proc, unsigned port_and_offset,
                 Word data1, Word data2, Word data3 Word data4)
188   void SendXferBothLRPollBoth(int proc, unsigned port_and_offset,
                 Word data1, Word data2, Word data3 Word data4)
188   void SendXferBothRLPollRight(int proc, unsigned port_and_offset,
                 Word data1, Word data2, Word data3 Word data4)
188   void SendXferBothLRPollRight(int proc, unsigned port_and_offset,
                 Word data1, Word data2, Word data3 Word data4)

186   HANDLER void StrataXferHeaderHandler(int port_num, int size,
                 void (*handler)())
188   HANDLER void StrataXferPut3Handler(unsigned port_and_offset,
                 Word data1, Word data2, Word data3)
188   HANDLER void StrataXferPut2Handler(unsigned port_and_offset,
                 Word data1, Word data2)
188   HANDLER void StrataXferPut1Handler(unsigned port_and_offset, Word data1
```

Multi-Block Transfers

```
188   ABXid AsyncBlockXfer(int proc, unsigned port, int offset, Word *buffer,
                 int size, void (*complete)(ABXid id, Word *buffer))
```

189	`void ServiceAllTransfers(int rounds)`
189	`void CompleteTransfer(ABXid id)`
189	`int GetABXproc(ABXid id)`
189	`unsigned GetABXport(ABXid id)`
189	`int GetABXoffset(ABXid id)`
189	`Word *GetABXbuffer(ABXid id)`
189	`int GetABXremaining(ABXid id)`
	Debugging
189	`ATOMIC int Qprintf(const char *format, ...)`
189	`ATOMIC int Qfprintf(FILE *out, const char *format, ...)`
189	`int EmptyPrintQ(void)`
190	**Macro** `assert(Bool x)`
190	`ATOMIC void StrataLog(const char *format, ...)`
	Graphics
192	`void InitTrace(void)`
192	`ATOMIC int SetState(int state)`
193	`Bool RecordStates(Bool on)`
193	`void OutputStateLog(void)`