# A Systematic Characterization of Application Sensitivity to Network Performance

by

Richard Paul Martin

B.A. (Rutgers University) May 1992
M.S. (University of California at Berkeley) December 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David E. Culler, Chair
Professor David A. Patterson
Professor Hal R. Varian

Spring 1999

The dissertation of Richard Paul Martin is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 1999

**A Systematic Characterization of**
**Application Sensitivity to Network Performance**

Copyright Spring 1999
by
Richard Paul Martin

## Abstract

A Systematic Characterization of
Application Sensitivity to Network Performance

by

Richard Paul Martin
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David E. Culler, Chair

This thesis provides a systematic study of application sensitivity to network performance. Our aim is to investigate the impact of communication performance on real applications. Using the LogGP model as an abstract framework, we set out to understand which aspects of communication performance are most important. The focus of our investigation thus centers on a quantification of the sensitivity of applications to the parameters of the LogGP model: network latency, software overhead, per-message and per-byte bandwidth. We define sensitivity as the change in some application performance metric, such as run time or updates per second, as a function of the LogGP parameters. The strong association of the LogGP model with real machine components allows us to draw architectural conclusions from the measured sensitivity curves as well.

The basic methodology to measure sensitivity is simple. First, we build a networking apparatus whose parameters are adjustable according to the LogGP model. To build such an apparatus we start with a higher performance system than what is generally available and add controllable delays to it. Next, the apparatus must be calibrated to make sure the parameters can be accurately controlled according to the model. The calibration also yields the useful range of LogGP parameters we can consider.

Once we have a calibrated apparatus, we run real applications in a network with controllable performance characteristics. We vary each LogGP parameter in turn to observe the sensitivity of the application relative to a single parameter. Sensitive applications will exhibit a high rate of "slowdown" as we scale a given parameter. Insensitive applications will show little or no difference in performance as we change the parameters. In addition, we can categorize the shape of the

slowdown curve because our apparatus allows us to observe plateaus or other discontinuities. In all cases, we must compare our measured results against analytic models of the applications. The analytic models serve a check against our measured data. Points where the data and model deviate from one another expose areas that warrant further investigation.

We use three distinct application suites in order to broaden the applicability of our results. The first suite consists of parallel programs designed for low-overhead Massively Parallel Processors (MPPs) and Networks of Workstations (NOWs). The second suite is a sub-set of the NAS parallel benchmarks, which were designed on older MPPs. The final suite consists of the SPECsfs benchmark, which is designed to measure Network File System (NFS) performance over local area networks.

Our results show that applications display the strongest sensitivity to software overhead, slowing down by as much as a factor of 50 when overhead is increased by a factor of 20. Even lightly communicating applications can suffer a factor of 3-5 slowdown. Frequently communicating applications also display strong sensitivity to various bandwidths, suggesting that communication phases are bursty and limited by the rate at which messages can be injected into the network. We found that simple models are able to predict sensitivity to the software overhead and bandwidth parameters for most of our applications. We also found that queuing theoretic models of NFS servers are useful in understanding the performance of industry published SPECsfs benchmark results.

The effect of added latency is qualitatively different from the effect of added overhead and bandwidth. Further, the effects are harder to predict because they are more dependent on application structure. For our measured applications, the sensitivity to overhead and various bandwidths is much stronger than sensitivity to latency. We found that this result stemmed from programmers who are quite adept at using latency tolerating techniques such as pipelining, overlapping, batching and caching. However, many of these techniques are still sensitive to software overhead and bandwidth. Thus, efforts in improving software overhead, per-message and per-byte bandwidth, as opposed to network transit latency, will result in the largest performance improvements across a wide class of applications demonstrating diverse architectural requirements.

We conclude that computer systems are complex enough to warrant our perturbation based methodology, and speculate how the methodology might be applied to other computer systems areas. We also conclude that without either much more aggressive hardware support or the acceptance of radical new protocols, software overheads will continue to limit communication performance.

Professor David E. Culler
Dissertation Committee Chair

to Mimi L. Phan.

## Acknowledgements

I must first thank my uncle, Dr. James Martin. Without his singular insight on the nature of graduate studies I would not have finished this thesis.

I must also thank my advisor, Dr. David E. Culler, for innumerable interesting conversations over the years. However, I am especially grateful for the guidance I received during those first critical years in graduate school. Your early sheparding on the HPAM system got me off to a great start.

My parents, Dr. Richard and Elma Martin, deserve much credit. Thanks for all the support you have provided, especially for all those computers. More importantly, thanks for sitting through my endless small demos. I must also thank my father for the "Professor X" book, which still gives me a good laugh.

To my long time fiance and now wife, Mimi Phan, I must thank for giving me the love, support and perspective to keep going during the long and difficult years.

I am greatly indebted to all the co-authors on various works, and what a list it has become. I'm honored to even have my name on the same pages as you: Amin Vahdat, Marc Fiuczynski, Lok Tin Liu, Remzi Arpaci-Dusseau, Frederick C.B. Wong, Chad Yoshikawa, Andrea Arpaci-Dusseau, Klaus Schauser, Randy Wang and Arvind Krishnamurthy.

I would also like to express thanks to my office-mates in 445 Evans, 467 and 466 Soda: Steve Luna, Cedric Krumbein, Brent Chun, Tony Chan, Vikram Makhija and Matt Welsh. You were a great group to work with and provided quite a bit of respite from the daily grind of graduate studies.

To the other members of the NOW project, thank you for six years simulating conversations. Predictions aside, most of us made it, and that's success enough. Thanks to Steve R., Doug, Kristin, Jeanna, Drew, Mike, Satoshi, Nisha, Kim, Eric, Alan, and Steve L. for making such a big project bearable.

Finally, I would like to thank my other two committee members, Dave Patterson and Hal Varian, for giving excellent advice during my quals talk as well as for reading this work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

*Nature is so complex and random that it can only be approached with a systematic tool that presupposes certain facts about it. Without such a pattern it would be impossible to find an answer to questions even as simple as 'What am I looking at?'* — James Burke, The Day the Universe Changed.

In recent years networking infrastructure and applications have experienced rapid growth. In the System Area Network (SAN) context designs are quite diverse, spanning a range of both hardware [16, 17, 23, 47, 53] as well as specialized system software [72, 81, 88, 103]. Local Area Networks (LANs) have also made impressive improvements, advancing from shared 10 Megabit (Mb) designs, to 100 and 155 Mb designs [15, 33], to switched gigabit designs [98, 99]. Wide Area Networks (WANs) are also experiencing equally dynamic growth: emerging systems include packet radio, Integrated Services Digital Network (ISDN), Cable Modems, Asymmetric Digital Subscriber Loop (ADSL) and Direct Broadcast Satellite (DBS). On the applications side, there are many new applications spanning these domains, such as protocol validators and network infrastructure services [41, 97].

The explosion of designs in networking technologies and applications has resulted in an increased demand for a systematic, quantitative framework in which to reason about performance. Historically, networking hardware and system software are evaluated using micro-benchmarks. Tools such as ttcp [22], lmbench [74], and netperf [58] provide the two most commonly used micro-benchmarks: Round Trip Time (RTT) and peak bandwidth (PB). Indeed, the evaluation of many designs [16, 33, 103] often only reports these two metrics. A third common metric is bandwidth as a function of transfer size. However, even this additional metric tells us very little about the underlying communication sub-system. Like the MIPS of CPU performance, micro-benchmarks in isolation provide little insight as to how improvements in networks improve application performance.

The exponential growth in CPU performance illustrates the benefit of a good conceptual framework that captures the essential performance tradeoffs of a system. An enormous improvement over the simple MIPS micro-benchmark was the “iron triangle”. [1] Although conceptually simple, the iron triangle model of application execution captures the essence of many performance tradeoffs without drowning in details. It gave a diverse community of application developers, compiler writers and computer architects a common framework in which to reason about the performance impact of different designs. The model thus allowed quantitative comparisons of processor designs using real programs. Indeed, a consistent theme across a variety of computer systems areas has been that the metrics and conceptual models used in evaluation are as important, if not more so, than the designs themselves.

This work uses an existing model, LogGP [2, 29], as a generalized framework for understanding application performance as it relates to the network hardware/software combination. The parameters of the model: latency, overhead, gap and peak bandwidth for large transfers, correspond well to the networking hardware and software components of real machines. An important feature of the LogGP model is that it allows application developers and machine designers to reason about the overlap between communication and computation. Thus, algorithms and applications can be characterized as to their degree of latency tolerance. The RTT and peak bandwidth metrics bundle overhead and latency into single parameters; characterizing an application’s tolerance to latency as distinct from overhead is impossible with those parameters alone. In addition, RTT and PB fail to capture important characteristics of distributed systems, which transfer “medium” sized data (in the single KB range) [106]. Although there are other algorithmic models in the literature [18, 43, 46, 101] which can capture some aspects of the network, they have been developed for parallel program design and so do not correspond well to machine components. Evaluating the components of the networking system as they relate to the application is easier with the LogGP model.

Once we have a firm grounding in a model, we can begin to make quantitative claims about networked computer systems in the language of the model, in our case LogGP. This thesis provides such a systematic study of the LogGP parameter space. Our aim is to investigate the impact of communication performance on real applications. Furthermore, we want to understand which aspects of communication performance are most important. We begin our investigation by quantifying the sensitivity of applications to the parameters of the LogGP model. We define sensitivity as some ap-

[1] The iron triangle is the model: CPU time = Instruction Count $\times$ Cycles per Instruction $\times$ Cycle Time.

plication change in performance, such as run time or updates per second, as a function of the LogGP parameters.

The methodology to measure sensitivity is conceptually simple, although somewhat involved in practice. First, we build a networking apparatus whose parameters are adjustable according to the LogGP model. To build such an apparatus we start with a higher performance system than what is generally available and add controllable delays to it. Next, we calibrate the apparatus to make sure we can control the parameters accurately.

Once we have a calibrated apparatus, we can run the applications in a network with the desired performance characteristics. We “turn the knob”, varying each parameter in turn and observing the resulting sensitivity of the application as a function of a single parameter. In all cases, we must compare our measured results against an analytic model of the application. The analytic models serve a check against our measured data. Points where the data and model deviate from one another expose potential anomalies that warrant further investigation.

After we have collected the sensitivity data, we can answer a range of questions. These include questions about the accuracy and applicability of our method, questions about application behavior, questions about communication architectures, and finally, questions related to the accuracy and applicability of simple application models.

The rest of this chapter is organized as follows. We first introduce two axes of experiment design fundamental to networking research. The first axis categorizes the nature of the questions of the experiment, and the second axis categorizes the nature of the experiment method. Next, we describe the contributions of this thesis. Finally, we present a roadmap of the thesis organization.

## 1.1 Background

What does a “better” network mean? In order to answer the question in a concrete manner a researcher must run quantitative experiments. As with any scientific experiment, a network experiment aims to elucidate the nature of the system’s response to some stimulus. In computer networking, the dependent variables are few, e.g., latency, bandwidth, packets-per-second, and run-time. The number and types of independent variables are many. Because the most common type of experiment evaluates two or more designs, the independent variable is often some aspect of the network design, such as a retry protocol or routing algorithm. The number of independent variables thus spans the entire design space. In addition to different types of variables, there are a number of methods used to carry out the experiments.

In this section we present a simple way to categorize a wide range of networking research. Our framework for categorizing computer network experiment design divides the design space along two axes. These axes provide intuition into the nature of this thesis' experiments, and taxonomic placement of this work into a broad spectrum of computer networking research.

The first axis categorizes the response of interest along the spectrum from network to application. Specifically, the axis defines the class to which the dependent variables belong. For example, run time clearly belongs to the application class. Packet latency across a set of routers belongs to the network class. Most experiments fall clearly into one class or another. Defining experiments along this axis is important because it dictates what can be abstracted. For example, if the network is the focus of the experiment, many application details can be eliminated. Likewise, if run-time is the dependent variable, we may wish to focus our efforts on understanding the application and abstract away network details.

The second axis categorizes the experimental method. Once the viewpoint of the experiment has been established (network or application), we have a number of methods of evaluation, e.g. simulation or direct measurement. These methods fall under general performance evaluation techniques; we survey them in order to gain insight as to the rational for the method chosen in this study.

In the next sections, we explore these two axes in greater detail. We start by describing the two points on the experiment paradigm axis. We then describe four methodologies used in networking experiments. Finally, we summarize with an overview of the strengths and weaknesses of the different experiment methods.

### 1.1.1 Experiment Paradigm Axis

Networking research traditionally has taken two perspectives with regards to evaluation. In any network evaluation, there is the network itself, as well as the application load placed on it. A network with no load is like a highway without cars—evaluation of the roadways is difficult if we do not understand the characteristics of the traffic load. Over-extending the analogy a bit, applications, like traffic, respond differently to changing network (road) conditions. In order to perform a tractable study, an experimenter often has to focus on one response or the other. That is, either the network-level or the application-level response will be the focus of the study.

### Network Focus

The first, and most common perspective, is to observe the network response to some application load. Two classic studies which exemplify this view are [24] and [56]. In these works, the application load is presented as a “black-box”: only a series of packets bound for different destinations is observable. The experiments use network-centric dependent variables, e.g. latency and bandwidth, as the metric of evaluation. The network perspective evolved because it is close to what is observable at network switches. All the switch “sees” is a stream of packets and traditionally they have little, if any, information on the applications’ characteristic load.

Because the network perspective contains such little information on application characteristics, many studies take a simple approach and model the applications as independent and identically distributed (IID) processes. These traditional loads have the twin advantages in that they are amenable to analytic analysis and are easy to correctly generate in a simulator. However, a serious drawback of these application models is that they do not approximate a real application load very well. A recent half-decade series of empirical studies [62, 66, 67], has shown that application traffic is bursty on all time scales. Traditional IID processes fail to capture this effect; the ramifications of this discrepancy are still under investigation.

### Application Focus

The second school of thought looks at the application performance given variable network parameters. In these classes of experiments, the dependent variables are application related. For example, what bandwidth can a single FTP transfer obtain with a certain loss rate?

The primary advantage of such a perspective is that it more closely models what a real user might think of as “better”. The computer architecture community has taken this approach to the most extreme level; every new microprocessor feature in the last 10 years has been defined by its impact on the SPEC benchmark suite. In the parallel architecture community, a few application suites have emerged, such as the SPLASH suite [110] and the NAS Parallel Benchmarks (NPB) [9, 11]. Among wider-area networks, the state of affairs is such that no real application suites have been defined for a broad segment of the community.

The application based perspective has the disadvantage of being quite complex to analyze. Unless we have a good models and an understanding of the applications, they can become opaque “black-boxes”. Experiments designed along this axis may not yield an understanding of how system design facets shape application behavior. A more difficult problem is that the application focus requires the researcher to map application-level observations to specific architectural features. An example of this problem is the lack of continuity across network interface designs. Returning to the processor architecture analogy, the SPEC benchmarks are the most analyzed programs on the planet, thus much is understood about their structure. A thorough analysis has been done for the SPLASH benchmarks [110], but little analysis has been done for the other benchmarks.

Assuming we cross the application complexity barrier, a second advantage of the application paradigm is that it focuses attention on the most important aspects of the network. For example, in the CPU world many programs are known to be quite sensitive to caches and branch predictors. The effect of this conclusion has been that many CPU designers focus on these aspects of their designs. With networks, latency or bandwidth alone are often touted as the figure of merit in many network-centric studies. In the WAN context, however, there are no agreed-upon standard application workloads. Without a deconstructed workload, a credible evaluation of new designs in the WAN arena will remain difficult.

### 1.1.2 Evaluation Methods Axis

Once the experimenter has chosen a perspective, a range of evaluation methodologies are available to perform the experiment. We classify them in order of increasing approximation to real systems. The four methods are: analytic modeling, simulation, emulation, and direct measurement. Each approach has appropriate uses and a range of strengths and weaknesses. The discussion here is presented as background to put this thesis into an appropriate context.

#### Analytic Modeling

Analytic modeling is traditionally not considered an “experimental” technique, although it could be classified as a gedanken experiment [2]. The purpose of an analytic model is often the same as an experiment, that is, answer basic questions about a network system, e.g., what is the peak bandwidth through the system? The difference is that the technique results in a set of equations that establish the relationship among various parameters and performance metrics. Unlike any of the other methods, nothing is “run” in the usual sense.

The complexity of such models can range from very simple frequency-cost pairs, to queuing theory models [50, 65], to advanced queuing theoretic techniques [57, 63]. Both the complexity and accuracy of such models can cover a large range, depending on the purpose of the model. Of-

[2] a thought experiment

ten a high degree of accuracy is not required, so analytic modeling is a viable technique in many circumstances.

Although at first they appear quite flexible, analytic models tend not to capture a wide range of system behavior because of the assumptions needed to make the equations tractable, and also to close the system of equations in the first place. For example, feedback loops in real systems are difficult to capture using analytic models. Chapter 5 presents observations describing the difference between measured data and models due to these effects.

In general, a scientific model is only "valid" as to accuracy of the model compared to real systems. For example, claims are often made in the popular press that Newtonian physics models are invalid because they can be quite inaccurate. However, a more accurate statement is that Newtonian mechanics are only valid in the realm of everyday experience. Questions regarding accuracy in the network models is a realm where much previous work in analytic modeling fails, not because of inaccurate models, but because of the lack of validation against actual systems. Indeed, general rules of thumb describing the accuracy of analytic models do not seem to exist for many classes of networks. This is a regrettable state of affairs because analytic modeling can be a powerful predictor of real systems.

An often overlooked power of analytic modeling is its ability to formalize how a system behaves, not in questions of absolute accuracy. That is, analytic models help us conceptualize the essence of the system, as opposed to merely describing the system's response to some stimulus. At this level, the value of the model is not in answering the absolute contribution of each component, but instead identifying the important components and their relationships in the first place. At some level, every researcher has a model in mind when answering performance analysis questions. The model's abstractions of a real system capture the essential elements without burdensome details. In addition, the formulation of a formal model provides the researcher a way to put these abstractions into a quantitative, testable form. The resulting advances in conceptual thinking are often the most valuable part of a model. For example, the simple analytic models described by the iron triangle in CPU design silenced endless artistic debates and forced designers into a realm of quantitative architectural tradeoffs. The absolute accuracy of such simple models for use in real programs was not addressed until much later [90].

**Simulation**

A step closer to a real system is a simulator. One might think of a simulator as a model without a closed form description. Simulation is perhaps the most popular technique in the research arsenal because of its infinite flexibility. A simulation experiment can potentially explore the entire design space. However, simulation suffers from a number of drawbacks some of which are commonly known, but a number of which are more subtle.

A well known drawback of simulation is its lack of scale. That is, the size of the system which can be simulated with any fidelity is often quite limited. Indeed, expanding the scale of simulation is an active area of research [54, 87]. In the parallel architecture regime, the effects of simulation scale have been studied extensively [110]. A primary effect is that architectural parameters must be scaled with the input in order to obtain meaningful results.

Simulation experiments also have a number of well known engineering drawbacks including complexity, time to build the simulator, time to debug the simulator, and time to validate the simulator. The importance of the last point cannot be over stressed. All too often, detailed simulator results are constructed without a corresponding validation either using an analytic model or a live system. While intuition can be used to validate a simulator [57], such an approach might only serve to feed the experimenter's previous bias.

An insidious danger with simulation is that the bias or misconceptions of the experimenter will become encoded in the simulator. While also true of analytic modeling, the danger is increased in simulation because of the increased level of detail when using simulation. Often, these bias manifest themselves as invalid assumptions. For example, the assumption that traffic has uniform random destinations is quite common and may not be true for many real networks [76]. In another example, one simulation study showed that paging over a network would be much faster than to local disk [5], but later results showed the performance to be much less than expected [3]. The inaccuracy originated because of the faulty assumption that network paging would have much lower overhead than disk paging.

A more subtle danger than faulty assumptions is that often the simulator is adept at simulating one class of system over another. For example, shared memory may be given preference over message passing because extensive simulation tools have been developed for shared-memory systems. The network experimenter may observe artifacts of shared memory machines, e.g. very short packets, while missing the effects of longer packets that occur when using message passing or WAN protocols.

### Emulation

We define emulation, as distinct from simulation, as when some part of a live system is run and the components which comprise the independent variables are emulated. That is, the part of the system under test is a real, while the parts which compose the independent component of the experiment are emulated. A primary advantage of such an approach is that it can be scaled larger than simulation. For example, the data set sizes in [73] are a factor of 10 larger than the simulation results for similar programs found in [52, 110].

Emulation is certainly not as flexible as simulation. A substantial limitation of emulation is that it misses at least an order of magnitude of the design space possible compared to simulation because the live system component requires real hardware and software. While the emulated portion can be modified, the interfaces between it and the live portion will dictate how much of the design space can be controlled. With full-blown simulation, only programming effort and computer resources limit the simulation.

Another drawback of emulation over simulation is the reduced fidelity of the experiments. It is often difficult to observe the live parts of the system in order to understand observed responses. In the operating system community, there has been much recent research on live system measurement [4]. An interesting tradeoff between fidelity and scale was explored by mixing multiple levels of simulation and emulation in [89]. In the programming language realm, binary translators have a similar effect [27, 92]. Increasing the fidelity of the "emulation" adds instructions, and this overhead can be adjusted according to the needs of the experimenter.

### Direct Measurement

Direct measurement is by definition the most "realistic" of experimental techniques. We define direct measurement as when only a complete system is used in the experiment; no part of the system is emulated or simulated. Most often, direct measurement is used to judge different systems in similar points in the design space. For example, [49] compares Autonet, Ethernet and FDDI.

Direct measurement can only evaluate discrete points in the design space rather than a range of designs. The single-point nature of direct measurement can be both beneficial and detrimental. On one hand, limiting the design space to a few points limits the number of independent variables. [3] In addition, since the measured systems are real we could actually use such systems. On the other hand, the inability to scale parameters in a controlled fashion makes determining impor-

[3] Section 2.4 backgrounds experiment factor design.

tant parameters in the design space difficult. Direct measurement does not help us conceptualize a system, and can even leave us more befuddled than before. The lack of a model may thus effect our conclusions. Because the system under measurement is an actual artifact, as opposed to a model, many studies often do not articulate the results in terms of abstract parameters.

A potential pitfall of using live systems is that the systems may not be configured "properly". For example, a version of Solaris had certain TCP window constants set too low as shipped from the manufacturer [61]. Although analyzing a system "as shipped" might still be a fair characterization, it does not represent the true potential of a mis-configured system. Validating the results of live systems against analytic models can expose mis-configuration errors.

### 1.1.3 Summary

Our two axes of categorization, experiment paradigm and methodology, allow us make sense of the broad field of network experiments. Network-centric studies describe what happens to the network, are useful for improving network performance, but tell us little about which aspects of the network are most important. Application-centric studies describe application behavior, point to network areas that yield the largest benefit, but do not describe how to make improvements.

On the methodology axis, we saw that analytic modeling requires little engineering effort and helps conceptualize how a system works. However, the models are somewhat inflexible in their ability to capture how real systems operate and thus can be inaccurate. Simulation requires substantial engineering, is limited in scale, but has unlimited flexibility. Emulation has limited flexibility because it requires live system components. It also entails substantial engineering effort. The results, however, can be quite accurate, because many important behaviors only appear at realistic input set sizes. Direct measurement is the most inflexible technique, does not help us conceptualize the system at all, but provides the most realistic results.

In the context of our framework, this thesis is clearly an application-centric, emulation based study. Our method allows us to reason about application behavior, and thus conclude about which aspects of the network are most important. Our emulation-based approach allows us to use a wide range of applications with realistic data sets and networking parameters. The live experiment data, coupled with analytic models, gives us greater confidence in our results and conclusions than using a single method.

## 1.2 Contributions

Our contributions fall into four areas. The primary contribution is an investigation of a novel performance analysis technique for network experimentation. Recall that because our study is application-centric, we can also make statements about application behavior. Answers to networking architecture questions make up the third area because of the close association of the LogGP model to real machine components. Modeling is our fourth, and smallest, area of contribution. Although the models are relatively simple, we can still draw some conclusions from them. The next sections describe this thesis's contributions in each of these areas. We also describe the questions this thesis attempts to answer in these areas, along with a brief summary of our results. Chapter 7 contains our humble attempt to answer these questions in more detail.

### 1.2.1 Performance Analysis

The primary contribution of this thesis is a methodology for the systematic exploration of the space of network design. The experimental methodology is inspired by analytic *bottleneck analysis* [57, 65], but is performed on a live system rather than on analytic models. The basic tenet of the approach is to introduce carefully controlled delays in key system components. The delays allow us to quantify the *sensitivity* of the system to each component. In our definition, sensitivity is the change in an application-centric metric as a function of one of the LogGP parameters. In contrast to bottleneck analysis, which only identifies peak performance, our method can expose bottlenecks as well as characterize their nature by examining the sensitivity curve.

Returning to our taxonometric experiment design space described in Section 1.1, our experimental method is application-centric and emulation based. The apparatus allows us to "sandbox" running applications in a network with the desired performance characteristics. That is, we run real applications in a controllable network. The novelty of our approach is the use of this artificial network. We gain the benefit of observing the effects that occur in applications running realistic data sets. The disadvantage of our approach is that, although we do use a network which is higher performance than what is generally available, we are unable to observe sensitivities beyond its operational limits. Another encumbrance of our method is that in order to understand the sensitivity curves, we must create models of the applications.

In spite of our method's shortcomings, it allows us good observational fidelity. Sensitive applications will exhibit a high rate of "slowdown" as we scale a given LogGP parameter. Insensitive applications will show little or no difference in performance as we change the parameters. Another important advantage of our method over traditional ones is that we can categorize the shape of the slowdown curve because our apparatus allows us to observe plateaus or other discontinuities. These discontinuities are of prime importance to the system architect because designs can take advantage of performance plateaus.

We find that our application-centric, emulation approach works well in practice. However, care must be taken to calibrate the apparatus, as well as cross-check the results against simple models. We are able to quantify the sensitivity curves of many different applications in a variety of programming domains, including parallel computing and distributed file systems. Our apparatus scales over an order of magnitude in overhead, latency and bandwidth. We can thus emulate networks ranging from high-performance SANs, to traditional LANs, and even WAN links.

### 1.2.2 Application Behavior

A unique approach of this thesis is the use of live systems in a bottleneck-style of analysis, as opposed to the more traditional approaches using analytic models. We find that a modern computer system is complex enough that our approach is a valid method of simply characterizing the application space.

The use of a live system allows us to observe and quantify real application behavior and validate models. Modeling results are always suspect until validated by other means. Although measurement as a technique can certainly quantify application behavior, using measurement alone it is difficult to obtain, without factor analysis, the sensitivity results provided for by our method.

We can obtain answers to a broad range of questions about application behavior using the methods in this thesis. The sensitivities to $L$,$o$,$g$, and $G$ are the most obvious. However, we can also ask questions such as is communication bursty, or uniform? Do applications contain many serial dependencies? If so, does the nature of these dependencies result in more sensitivity to overhead or latency? We can also begin to explore questions such as if we can we restructure the application to better suit modern machine characteristics. If we observe enough applications, we can also hope to make some generalizations about how programmers use these systems.

We find that programmers employ a diverse set of latency tolerating techniques and that these are quite effective in practice. However, many of these techniques are still sensitive to overhead. Thus, many applications demonstrated the strongest sensitivity to software overhead. For both a range of parallel programs and NFS, overhead is a limiting factor. Continued improvements in software overhead will yield the largest benefits. Applications also demonstrated strong sensitivity to various bandwidths, suggesting that communication phases are bursty and limited by the rate at which messages or bytes can be injected into the network. Examining the history of many programs, we see that program optimizations shift sensitivity away from latency and overhead and towards bandwidth.

### 1.2.3 Network Architecture

Armed with the sensitivity results and the LogGP model, we can answer a number of important architectural questions. For example, how much do network design changes affect application performance? How does increasing network performance improve application performance? Would additional hardware support be worth the costs? If so, what architectural features would yield the largest benefit?

LogGP plays a key role in our ability to answer these architectural questions because the parameters of the model were designed to correspond to real machine components. The LogGP model has been validated as an architectural model, as well as a parallel programming model, on a wide variety of real machines and message layers [31, 55]. We can thus draw strong conclusions about how improvements in machines will affect application performance based on our abstract sensitivity results. For example, if we improved the networking code by a factor of 2, and if our sensitivity models for $o$ have a slope of 1 (i.e. each factor of $o$ improves performance by 1 unit) then we can have reasonable confidence that our improved networking code would result in an overall $2\times$ performance improvement.

Our result that applications demonstrate the strongest sensitivity to software overhead implies that continued improvements in software overhead will yield the largest benefits. Hardware and software should be designed to minimize software overhead, perhaps even at the expense of some of the other parameters. In particular, our results show the somewhat counter intuitive result that architectures that minimize overhead at the expense of network latency can deliver increased overall performance for most applications.

Applications also demonstrated strong sensitivity to various bandwidths. Although many designs can deliver near peak per-byte bandwidths for long messages, we also measured a number of applications where small message throughput was nearly as important. Architectures must deliver not only low overheads, but also should deliver close to the peak bandwidth for small messages. As with overhead, our results on application behavior show that architectures that can sustain a high per-message rate for small messages at the expense of latency can still maintain good overall application performance.

Chapter 6 explores a novel architectural designs based on the results of this thesis. We use SPINE [40], a software infrastructure, to examine the effect of overhead reduction by adding computation into the network device. The results, while encouraging, show that our understanding of overhead reduction techniques is still quite limited and more investigation in this area is warranted. We show that more radical software and hardware architectures will be needed to deliver order of magnitude performance improvements to applications while maintaining connectivity to standard protocols.

### 1.2.4 Modeling

Because we use analytic modeling to validate the results of our live system experiments, we can ask questions about the models themselves. For example, how well do simple models[4] predict application run time? How accurate are simple models compared to live systems? If they are not sufficient, how closely do fully characterized applications (e.g. those in [39]) match the known data?

Our simple models are by definition linear in event frequency and cost. If the measured sensitivities are linear to the parameters simple models may be sufficient. More complex characterizations such as those in [39] consider dependencies between communication events. A recent complex variant of the LogP model [43] considers queuing effects due to endpoint congestion. For this work we will use simple models wherever possible. We find that simple models can capture the high-level impact of changing network performance, particularly for overhead, but more complex models are needed when good fidelity is required.

We use a basic queuing theory to model NFS. While simple models may be useful in this case, servers traditionally have been described using queuing theoretic models. Our apparatus allows us to compare measured results against this class of traditional models. We find that queuing theory is able to capture much of the behavior of an NFS system. However, at high loads the feedback control loops in NFS cause the measured values to diverge from the models substantially.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes the methodology in detail. We describe our basic apparatus. We first background the basic communication software

[4]We define a **simple** model to be one were the execution time is a linear function of a series of $n$ event types where each type has a frequency $F_i$, and cost $C_i$. Thus, total execution time = $\sum_{i=1}^{n} F_i \times C_i$

and then describe how we added controllable delays to the different system components. Next, we provide an outline of the micro-benchmark calibration technique used to make sure the apparatus is operating correctly. We then document two variations of the basic apparatus, named after the communications layers built on top of the basic apparatus: MPI and TCP/IP. The next section in Chapter 2 describes the single parameter at-a-time approach we have chosen to use for our experiments, as opposed to some other combination of scaling the parameters. We then document alternative models to LogGP, followed by a discussion of why LogGP is the most appropriate choice for our work. We conclude Chapter 2 with a description of four related experiments an compare their methodologies to that of this thesis using the framework we developed in Section 1.1.

The next three chapters describe the sensitivity findings of our methodology for three application suites. Each chapter first characterizes the applications. Next, we present analytic models of the applications. We then present the sensitivity results along with a description of the utility of the models. Finally, we summarize the results of each suite.

Chapter 3 presents the results for a set of Split-C and Active Message programs. Our apparatus performed admirably, exposing sensitivity curves for all the parameters. The apparatus exposed a number of non-linearities in the sensitivity curves and found an interesting anomaly in one application. We find that these programs are very sensitive to overhead and gap, and quantitatively less sensitive to latency and Gap. Our linear models were good at predicting slowdowns due to increased overhead, were less successful at predicting the effects of gap, and rather poor at predicting the effects of increased latency and per-byte bandwidth.

Chapter 4 presents the sensitivities of some of the NAS Parallel Benchmarks (NPB). Communication patterns of the NPB are much different from the Split-C/AM programs. Communication is dominated by few, large messages. Unlike the Split-C/AM programs, the NPB are insensitive to overhead, gap and latency. They do, however, show some sensitivity to Gap. The primary network architectural feature relevant to these applications is network bisection bandwidth.

Chapter 5 presents the sensitivity results and models of Sun's Network File System (NFS). We find that like the Split-C programs, NFS is quite sensitive to added software overhead. NFS also displayed sensitivity to latencies in the LAN region of milliseconds, but showed a flat region for network latency under 150 microseconds.

In Chapter 6 we explore the SPINE system which applies some of the results of the previous chapters. One goal of SPINE is to improve performance by reducing overhead. In effect, SPINE reduces overhead by pushing some of the application work into the other parameters via intelligent network interfaces. The chapter briefly introduces a pipelining framework to better model the communication system, followed by the performance results of the actual system. We show that SPINE's performance improvements are mixed; it reduces overhead but does not improve the other parameters. The utility of the SPINE approach thus depends on the application context. We conclude the chapter with some thoughts on areas of future research.

Chapter 7 brings the results of the previous five chapters together. We organize the conclusions around the four areas of contributions: performance analysis, behavior, architecture and modeling. We hypothesize that our perturbation style of analysis could be used in a variety of computer systems contexts. The chapter also draws some parallels between the style used in this work and experiments in other sciences. We end the thesis with some final thoughts on the meaning of our findings.

# Chapter 2

# Methodology

*... it were far better never to think of investigating the truth at all, than to do so without a method.* — René Descartes, Rules for the Direction of the Mind.

In this section, we describe our experimental methodology. We first describe the design philosophy behind our method, followed by a placement of our experiments in the wider context of experiment design. Next, we describe the LogGP network model, whose parameters correspond directly to the factors in our experiments.

The bulk of this chapter describes the networking component of the apparatuses used in the experiments. The three systems are named after the primary communications layers used by the applications: Split-C/AM, MPI and TCP/IP. Aspects of the apparatus not related to communication, such as changes in the disk-subsystem, are described in later chapters.

After describing the apparatuses, we focus on our methodology for varying the network parameters. We use a single-factor at a time approach, scaling each LogGP factor in turn. We justify our use of single factor design by comparing other styles of experiment design, such as $2^k$ designs.

We conclude this chapter with a discussion of related network models and previous work. The purposed of the related models section is to describe why LogGP is an appropriate network model. We compare and contrast LogGP with other models developed in the literature. The examination of previous work focuses primarily on the experimental methods used. We place each work in the framework described in Section 1.1. The secondary purpose in examining previous work is to understand how their results compare or contrast with the results of this thesis. By examining a wide range of other experiments, we may be able to reinforce or diminish our results.

## 2.1 Experiment Design Philosophy

In the space of experimental design, this work uses application-centric metrics combined with an emulation methodology. The basic approach is to determine application sensitivity to machine communication characteristics by running a benchmark suite on a large system in which the communication layer has been modified to allow the latency, overhead, per-message bandwidth and per-byte bandwidth to be adjusted independently. This four-parameter characterization of communication performance is based on the LogGP model [2, 29], the framework for our systematic investigation of the communication design space. By adjusting these parameters, we can observe changes in the execution time or throughput of applications on a spectrum of systems ranging from the current high-performance clusters to conventional LAN based clusters.

We validate the emulation with analytic models. The models range from simple frequency-cost pairs to simple queuing networks. The intent of the models is to validate the emulation experiments. A side benefit of the models is that we can compare the accuracy of the models against live systems. The absolute accuracy can serve as a guide for future designers as to the applicability of analytic models to their situations.

In order to both demonstrate the soundness of the methodology, as well as draw general conclusions about application behavior, we must have a representative application suite. While no suite can possibly capture all application behavior, a diverse suite may capture the relevant structures of a broad class of programs. Our suite includes a variety of parallel programs written in the Split-C programming language, a sub-set of the NAS Parallel Benchmarks and the SPECsfs benchmark.

## 2.2 LogGP Network Model

When investigating trade-offs in communication architectures, it is important to recognize that the time per communication operation breaks down into portions that involve different machine resources: the processor, the network interface, and the actual network. However, it is also important that the communication cost model not be too deeply wedded to a specific machine implementation. The LogGP model [2, 29] provides an ideal abstraction by characterizing the performance of the key resources, but not their structure. A distributed-memory environment in which processors physically communicate by point-to-point messages is characterized by four parameters (illustrated in Figure 2.1).

$L$: the *latency*, or delay, incurred in communicating a message containing a small number of words

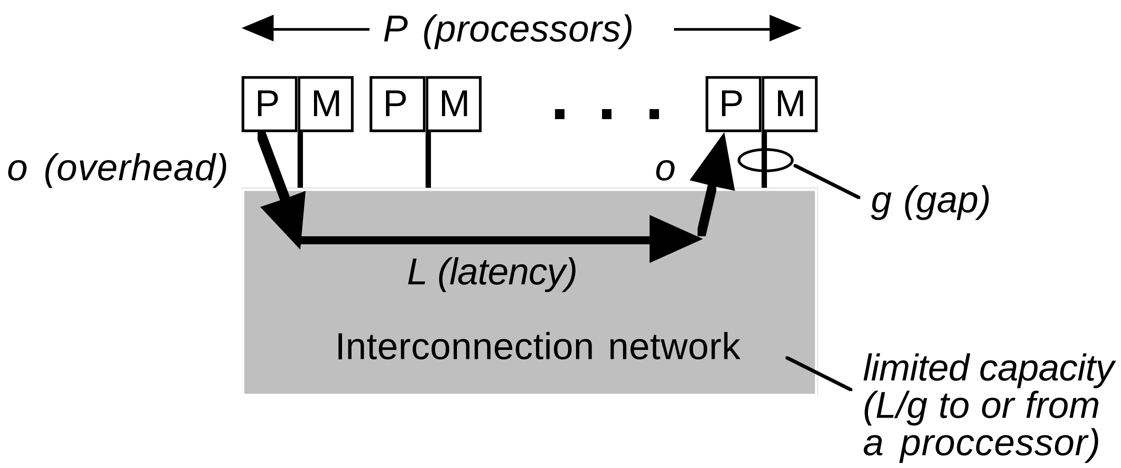


Figure 2.1: **LogGP Abstract Machine**
*The LogGP model describes an abstract configuration in terms of five performance parameters: $L$, the latency experienced in each communication event, $o$, the overhead experienced by the sending and receiving processors, $g$, the gap between successive sends or successive receives by a processor, $G$, the cost-per-byte for long transfers, and $P$, the number of processors/memory modules.*

from its source processor/memory module to its target.

$o$: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

$g$: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a module; this is the time it takes for a message to cross through the bandwidth bottleneck in the system.

$P$: the number of processor/memory modules.

$L$, $o$, and $g$ are specified in units of time. It is assumed that the network has a finite capacity, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit.

The simplest communication operation, sending a single packet from one machine to another, requires a time of $L + 2o$. Thus, the latency may include the time spent in the network interfaces and the actual transit time through the network, which are indistinguishable to the processor. A request-response operation, such as a read or blocking write, takes time $2L + 4o$. The processor

| Platform | $o$ ($\mu$s) | $g$ ($\mu$s) | $L$ ($\mu$s) | MB/s($\frac{1}{G}$) |
|---|---|---|---|---|
| Berkeley NOW | 2.9 | 5.8 | 5.0 | 38 |
| Intel Paragon | 1.8 | 7.6 | 6.5 | 141 |
| Meiko CS-2 | 1.7 | 13.6 | 7.5 | 47 |

Table 2.1: **Baseline LogGP Parameters.**
*This table shows the performance of the hardware platform used, the Berkeley NOW. Two popular parallel computers, the Intel Paragon and the Meiko CS-2 are included for comparison.*

issuing the request and the one serving the response both are involved for time $2o$. The remainder of the time can be overlapped with computation or sending additional messages.

The available per-processor message bandwidth, or communication rate (messages per unit time) is $1/g$. Depending on the machine, this limit might be imposed by the available network bandwidth or by other facets of the design. In many machines, the limit is imposed by the message processing rate of the network interface, rather than the network itself. Because many machines have separate mechanisms for long messages, e.g., DMA, it is useful to extend the model with an additional gap parameter, $G$, which specifies the time-per-byte, or the reciprocal of the bulk transfer bandwidth [2]. In our machine, $G$ is determined by the DMA rate to or from the network interface, rather than the network link bandwidth.

The LogGP characteristics for the Active Message layer are summarized in Table 2.1. For reference, we also provide measured LogGP characteristics for two tightly integrated parallel processors, the Intel Paragon and Meiko CS-2 [31].

## 2.3 Apparatuses

In this section we describe the various apparatuses used. The experimental apparatus consists of commercially available hardware and system software, augmented with publicly available research software that has been modified to conduct the experiment. There are three distinct variations of the same basic apparatus. All use the an Active Messages [104] variant called Generic Active Messages (GAM) [30]. The primary differentiator between the apparatuses is the high-level transport layered on top of this basic messaging substrate, Split-C [28], MPI [75] or TCP/IP. Each apparatus is used for one application suite: Split-C/AM for the Split-C applications, MPI-GAM for the NAS Parallel Benchmarks, and TCP-GAM for the SPECsfs benchmark. The Split-C and MPI apparatus use the identical GAM Active Message layer. The Active Message Layer for the TCP/IP version of the apparatus required substantial modification to the semantics of the Active Message layer.

### 2.3.1 Basic Split-C/AM Apparatus

In this section we first describe the hardware used. Next, we provide background on the GAM layer, which forms the core communication system of our apparatus. We then describe how we vary the LogGP parameters by engineering controllable delays into GAM. Finally, we briefly describe how we calibrated the apparatus using a simple microbenchmarking technique.

#### Hardware

The hardware for all our experiments is a NOW of 35 UltraSPARC Model 170 workstations (167 MHz, 64 MB memory, 512 KB L2 cache) running Solaris 2.5. Each has a single Myricom M2F network interface card on the SBUS, containing 128 KB SRAM card memory and a 37.5 MHz "LANai" processor [17]. The processor runs our custom firmware, called the LANai Control Program (LCP). The LANai processor plays a key role in allowing us to independently vary LogGP parameters. The machines are interconnected with ten 8-port Myrinet switches (model M2F, 160 MB/s per port) in a two-level fat tree topology. Each of the 7 switches in the first level is connected to five machines and all three second level switches. At any given time, we only run programs on 32 machines. Often a machine or two was down; a few spares went a long way towards having 32 working machines at any given time.

#### GAM Active Message Layer

The GAM Active Message layer on Myrinet was developed as an experimental research prototype. Its primary goal was to deliver high performance communication to parallel applications on NOWs. Although GAM is not strictly necessary for use in this study, two of its characteristics proved quite useful. First, its high performance increased the range of the LogGP parameter space we can consider. Second, its simplicity allowed for easy insertion of delays into various portions of the system.

The GAM Active Message layer follows a request-reply model. The underlying network is assumed to be reliable, but only possesses finite buffering. Because of the finite buffering, care must be taken to avoid fetch-deadlock. Deadlock avoidance is achieved by using credit counts between pairs of nodes. This is the $kP$ algorithm described in [32] and similar to the one used in [72]. The

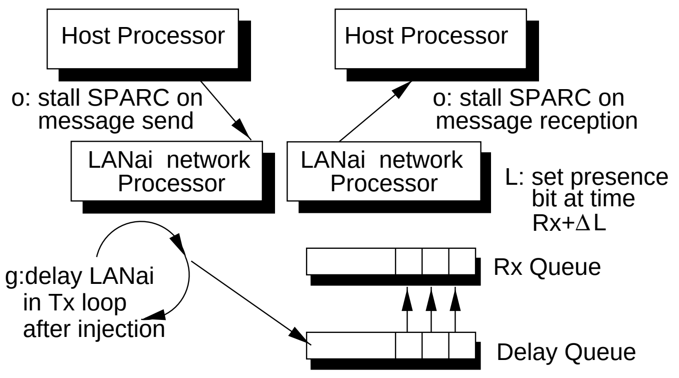

Host Processor
Host Processor
o: stall SPARC on message send
o: stall SPARC on message reception
LANai network Processor
LANai network Processor
L: set presence bit at time Rx+ΔL
g:delay LANai in Tx loop after injection
Rx Queue
Delay Queue

Figure 2.2: **Varying LogGP Parameters**
*This figure describes our methodology for individually varying each of the LogGP parameters. The interaction between the host processor, the network processor (LANai) and the network is shown for communication between two nodes.*

layer is not thread-safe and requires polling to receive messages. Polls are automatically inserted when sending messages, however.

In addition to requests and replies, messages are typed as short or long. Short messages are up to 6 words in length, with one word consumed as a function handler. Long messages contain a function pointer, two words for function arguments and a block of data up to 4KB long. Short and long messages are orthogonal to requests and replies. Thus, a short or long message may be sent in response to either type of request. A library function performs the packetization for direct memory-copy requests longer than 4KB. Note that in the GAM specification [30] there is not an arbitrarily long reply bulk-transfer function; replies in the Myrinet apparatus are limited to 4KB.

### Varying the LogGP Parameters

The key experimental innovation is to build adjustments into the communication layer so that it can emulate a system with arbitrary latency, overhead, gap and Gap. Our technique is depicted in Figure 2.2 which illustrates the interaction of the host processor, the LANai (network interface processor) and the network for communication between two nodes. The next sections describe how we varied each parameter in detail.

**Overhead** The majority of the overhead is the time spent writing the message into the network interface or reading it from the interface. Thus, varying the overhead, $o$, is straightforward. For each message send and before each message reception, the operation is modified to loop for a specific period of time before actually writing or reading the message.

**gap and Gap** The gap is dominated by the message handling loop within the network processor. Thus, to vary the gap, $g$, we insert a delay loop into the LCP message injection path after the message is transferred onto the wire and before it attempts to inject the next message. Since the stall is done after the message is actually sent, the network latency is unaffected. Also, since the host processor can write and read messages to or from the network interface at its normal speed, overhead should not be affected. We use two methods to prevent excessive network blocking from artificially affecting our results. First, the LANai is stalled at the source rather than the destination. Second, the firmware takes advantage of the LANai's dual hardware contexts; the receive context can continue even if the transmit context is stalled.

To adjust $G$, the transmit context stalls after injecting a fragment (up to 4KB) for a period of time proportional to the fragment size. We stall the LCP for an adjustable number of microseconds for each 100 bytes of up to a 4 KB fragment. For example, if the Gap "knob" was set to 11, we would stall the LANai transmit context for an extra 11 $\mu$s for each 100 bytes of data in a fragment.

**Latency** The latency, $L$, requires care to vary without affecting the other LogGP characteristics. It includes time spent in the network interface's injection path, the transfer time, and the receive path, so slowing either the send or receive path would increase $L$. However, modifying the send or receive path would have the side effect of increasing $g$. Our approach involves adding a delay queue inside the LANai. When a message is received, the LANai deposits the message into the normal receive queue, but defers setting the flag that would indicate the presence of the message to the application. The time that the message "would have" arrived in the face of increased latency is entered into a delay queue. The receive loop inside the LANai checks the delay queue for messages ready to be marked as valid in the standard receive queue. Modifying the effective arrival time in this fashion ensures that network latency can be increased without modifying $o$ or $g$.

### Calibration

With any empirical apparatus, as opposed to a discrete simulator, it is important to calibrate the actual effect of the settings of the input parameters. In this study, it is essential to verify that our technique for varying LogGP network characteristics satisfies two criteria: first, that the communication characteristics are varied by the intended amount and second that they can be varied independently.

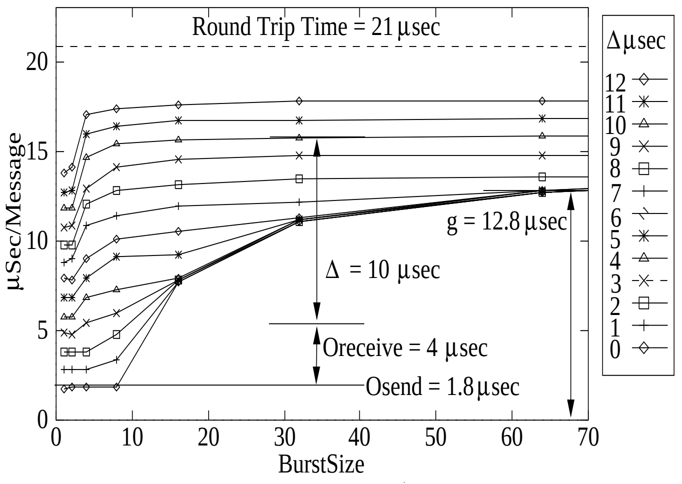


Figure 2.3: **Calibration of LogGP Parameters**
*The LogP* signature *is visible as the isobaric plot of burst size vs. fixed computational delay, $\Delta$. This signature was a calibration made when the desired $g$ was 14 $\mu$s. The send overhead, receive overhead and gap can be read from the signature. Overhead is modeled as the average of the send and receive overhead. Latency is computed as $\frac{1}{2}$(round trip time)-2o.*

Such a calibration can be obtained by running a set of Active Message micro-benchmarks, described in [31]. The basic technique is to measure the time to issue a sequence of $m$ messages with a fixed computational delay, $\Delta$ between messages. The clock stops when the last message is issued by the processor, regardless of how many requests or responses are in flight. Plotting the average message cost as a function of the sequence size (burst size) and added delay generates a LogP *signature*, such as that shown in Figure 2.3. Each curve in the figure shows the average initiation interval seen by the processor as a function of the number of messages in the burst, $m$, for a fixed $\Delta$. For a short sequence, this shows the send overhead. Long sequences approach the steady-state initiation interval, $g$. For sufficiently large $\Delta$ the bottleneck is the processor, so the steady state interval is the send overhead plus the receive overhead plus $\Delta$. Finally, subtracting the two overheads from half the round-trip time gives $L$.

Table 2.2 describes the result of this calibration process for three of the four communication characteristics. For each parameter, the table shows the desired and calibrated setting for that parameter. For much of the tunable range for overhead, the calibrated value is within 1% of the desired value. Observe that as $o$ is increased, the effective gap increases because the processor becomes the bottleneck, consistent with the LogGP model. As desired, the value of $L$ is independent of $o$. The calibrated $g$ is somewhat lower than intended and varying $g$ has little effect on $L$ and no effect on

| Desired | Observed | | | Desired | Observed | | | Desired | Observed | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| o | o | g | L | g | g | o | L | L | L | o | g |
| 2.9 | 2.9 | 5.8 | 5.0 | 5.8 | 5.8 | 2.9 | 5.0 | 5.0 | 5.0 | 2.9 | 5.8 |
| 4.9 | 5.1 | 10.1 | 5.0 | 8.0 | 7.5 | 2.9 | 5.1 | 7.5 | 8.1 | 2.9 | 6.3 |
| 7.9 | 8.1 | 16.0 | 4.7 | 10 | 9.6 | 2.9 | 5.5 | 10 | 10.3 | 2.9 | 6.4 |
| 12.9 | 13.0 | 26.0 | 5.0 | 15 | 14 | 3.0 | 5.5 | 15 | 15.5 | 2.9 | 7.0 |
| 22.9 | 23.1 | 46.0 | 4.9 | 30 | 29 | 3.0 | 5.5 | 30 | 30.4 | 2.9 | 9.6 |
| 52.9 | 52.9 | 106.0 | 5.4 | 55 | 52 | 2.9 | 5.5 | 55 | 55.9 | 3.0 | 15.5 |
| 77.9 | 76.5 | 151.0 | 5.3 | 80 | 76 | 2.9 | 5.5 | 80 | 80.4 | 2.9 | 21.6 |
| 102.9 | 103.0 | 205.9 | 6.0 | 105 | 99 | 3.0 | 5.5 | 105 | 105.5 | 3.0 | 27.7 |

Table 2.2: **Calibration Summary**
*This table demonstrates the calibration of desired LogP parameter values versus measured values. The table also shows that the LogP parameters can mostly be varied independent of one another. As predicted by the LogP model, when $o > g$ then $g$ is not longer observable as a distinct parameter, it degenerates to $o$. Note that in the steady-state, the data for $g$ includes the time to send* **and** *receive a message. The increase in $g$ at high $L$ is due to our system's fixed capacity of 4 outstanding messages between processor pairs. It differs from the LogP capacity model which specifies that up to $\lceil L/g \rceil$ messages can be in-flight to a given processor at a time.*

$o$. Increasing $L$ has little effect on $o$. A notable effect of our implementation is that for large values of $L$, the effective $g$ rises. Because the implementation has a fixed number of outstanding messages independent of $L$, due in part to the $kP$ deadlock avoidance algorithm, when $L$ becomes very large the implementation is unable to form an efficient network pipeline. In effect, the capacity constraint of our system is constant, instead of varying with $L$ and $g$ as the LogGP model would predict.

To calibrate $G$, we use a similar methodology, but instead send a burst of bulk messages, each with a fixed size. The delay inside the LANai bulk handling loop was set to a specific number of microseconds per 100 bytes of data. From the initiation interval and message size we derive the calibrated bandwidth. We increase the bulk message size until we no longer observe an increase in bandwidth, which happens at a 2K byte message size. Figure 2.4 shows a linear relationship between the added delay in the LANai code and the observed $G$. The linear relationship shows that the apparatus can deliver a range of $G$ quite accurately. The small dip at the lower left shows that as we add a linear delay, a sublinear increase in $G$ occurs. We can conclude from this probe that the baseline firmware is not rate-limited, instead the system is overhead-limited.

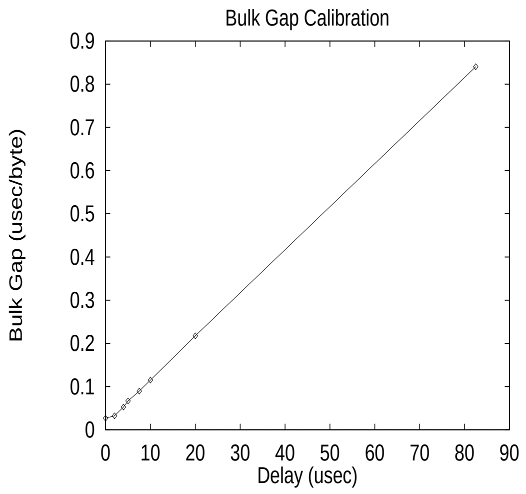


Figure 2.4: **Calibration of Bulk Gap for the Parallel Program Apparatus**
*This figure shows the empirical calibration for bulk Gap. The dependent variable shows the added delay in $\mu$s per 100 bytes of packet size. The independent variable is the Gap expressed in $\mu$s per byte (1/bandwidth) at a 2KB packet size. After a small delay, the relationship is linear, showing that the apparatus for adjusting bulk Gap is quite accurate.*

### 2.3.2 MPI Apparatus

The last few years has seen a standard message passing interface, aptly named the Message Passing Interface (MPI) [75], emerge from the parallel programming community. In this section, we describe the construction and performance of MPI on top of our basic apparatus described in the previous section. Recall that this apparatus is used in our study of the NAS Parallel Benchmark suite. We conclude this section with a simple model which describes how the MPI will react to changes in LogGP parameters.

The MPI specification in quite complex, including many collective operations, four semantic classes of point-to-point messages, methods of grouping processes (communicators), and many ways of tag-matching between sends and receives. In order to manage this complexity, the MPICH implementation [6] layers the more complex MPI abstractions on top of simpler ones. For example, collective operations, such as `MPI_All_to_all` are implemented as standard point-to-point messages using `MPI_send`. The point-to-point messages are in turn, mapped to the lowest layer, the MPI abstract device (MPID). The MPID layer is quite small; it implements just three ways to send point-to-point messages.

**Construction**

In order to construct a tunable apparatus, it was sufficient to map the MPID layer to GAM [109]. The three ways to send messages at the MPID level correspond to two mappings to the GAM level. The first send type, the “standard” send, is the most common. More importantly, of the three MPID sends, the standard send is the only one used by the NPB. An MPI layer standard send eventually maps to the MPID function `MPID_AM_Post_send`. This function sends a contiguous memory region to another processor, and returns when the data has been accepted by the communications layer. The tag and communications group are already specified by higher MPI layers. All MPI receive functions map to a single receive function at the MPID layer, `MPID_AM_Post_recv`. This function tests for completion at the processor that receives the message. For the standard send, no messages are sent inside the `MPID_AM_Post_recv` call.

The implementation strategy for the standard send depends on the size of the message sent. The GAM interface provides 2 distinct message sizes: 0 - 4KB, via the `am_request` function, and greater than 4KB in the `am_store` function. Each of the methods results in substantially different start-up costs and per-byte bandwidths, resulting in two methods of constructing sends at the MPID level.

The `MPID_AM_Post_send` call is mapped to the GAM layer using the Myrinet specific `am_request` function for messages less than 4KB long. The `am_request` function was added after the initial GAM specification in order to handle the “medium” message sizes needed by many distributed systems. It delivers a continuous block of data up to 4KB long, and invokes a handler on the remote end when the block arrives. The block on the remote side exists only for the life of the handler. For these medium message sizes, the MPID-GAM implementation simply launches the message into the network, or stalls if the network is full. Upon arrival, if the receive is posted, the data is copied into the final destination. If the receive has not been posted, the message is copied into a temporary buffer. Control messages, e.g., for barriers, are implemented with the `am_request_4` function. Recall that `am_request_4` sends an active message with 4 32-bit words as arguments.

For messages larger than 4KB, `MPID_AM_Post_send` first performs a round trip using the `am_request_4` call. The receiver returns the destination address of the location of the receive buffer. If the receive has not been posted, the handler on the receiver creates a temporary buffer. The sender blocks until it receives the response containing the address of the receive buffer. Once the address has been obtained, the sender uses the `am_store` function to send the data into the correct destination. Recall that `am_store` copies a block of arbitrary data from one node to another, and

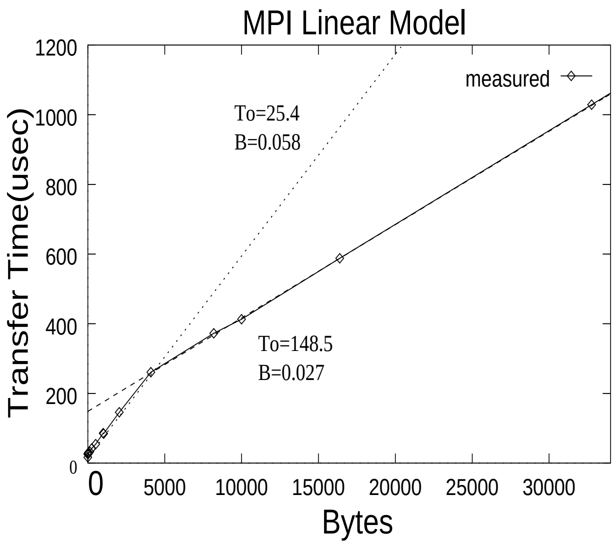


Figure 2.5: **Baseline MPI Performance**
*This figure shows the baseline performance of the MPI-GAM system. The figure plots half the round trip time for different size messages. Two distinct performance regimes are observable, one for messages $<$ 4KB and the other for messages $>$ 4KB. The modeled start-up cost, $T_0$ is obtained from the y-intercept of a least squares fits to the two performance regimes. The per-byte costs, $B = \frac{1}{R_\infty}$, is obtained from the slopes of the fitted lines.*

thus requires all memory addresses to be known in advance of the call. A key point of the GAM implementation is that `am_store` internally maps to a sequence of `am_request` calls. The GAM LCP can pipeline these requests resulting in the maximum bandwidth of 38 MB/s for a long sequence of 4KB `am_request` messages.

## Performance

In this section, we investigate the performance of the MPI message passing layer built on top of GAM. The purpose of this section is to understand how the inflation of the LogGP parameters at the GAM level affects the performance of MPI. We show how the different implementations of the standard MPI send result in different performance regimes.

MPI benchmarks traditionally use a linear model of performance. In the traditional linear model, a per message start-up cost of $T_0$ is paid on every message. A second parameter, $R_\infty$, captures bandwidth limitations of the machine. The cost to send an n-byte message, $T_n$, is thus modeled as $T_n = T_0 + \frac{n}{R_\infty}$. Fitting this model into the LogGP perspective requires care, although by definition, $G \equiv \frac{1}{R_\infty}$. Modeling $T_0$ requires knowledge of the underlying implementation. For example, $T_0$

may correspond directly to $o$, or, if a round trip is required, may include $L$ as well.

The classic experiment to compute $T_n$ is realized in the benchmark [35, 36]. This code performs a ping-pong of messages between two nodes. One node sends a message of size $n$ bytes. After the entire message is received, the second node responds with a message of size $n$ bytes. The total time for the test approximates $2 \times T_n$. The time reported is half of the time to send the message and receive the response.

Figure 2.5 shows the results of the experiment for the MPI-GAM system for increasing $n$. The first thing the note is that there is a sharp inflection point at 4KB. The slope of the line changes suddenly at 4 KB because of the change in the way MPI-GAM maps MPI messages to the GAM system. The figure clearly shows the tradeoff in increased start-up cost vs. delivered bandwidth. If we break up the line into to regions and compute the a least squares fit, we see that the fit is quite good for each of the two regions.

#### LogGP to MPI Model

Given that we know the basic MPI performance and protocol, we are now in a position to model the effect of inflation of the LogGP parameters on MPI performance. We characterize the change in parameters on the linear model described in the previous section.

Inflation of overhead will impact the system in three ways. The first and most obvious way is on $T_0$. For messages under 4 KB, the MPID to GAM protocol only uses one message, so we simply add $o$ to the cost of $T_0$. For messages over 4 KB, the protocol uses a round trip, and so in our model we inflate $T_0$ by $4o$. The third way overhead impacts the system is for long messages. We add $o$ to the cost of each 4 KB fragment, thus reducing the effective $R_\infty$. Adding latency to the system primarily impacts the $T_0$ term. We model an increase in $L$ as adding $2L$ to the cost of $T_0$ for messages over 4 KB and ignore effects for messages $\leq$ 4 KB. The gap is perhaps the most difficult to model. Because the MPI-GAM system uses few small messages, we chose to ignore added $g$ entirely. We shall see that is the not all that poor of an assumption. Indeed, one of the NPB can ignore added $g$ entirely. The Gap is perhaps easiest to cast into the MPI linear framework. Changes in Gap correspond directly to changes in $R_\infty$ via the model $R_\infty = \frac{1}{G}$.

### 2.3.3 TCP/IP Apparatus

The TCP/IP apparatus operates along the same lines as the parallel program apparatus. We use this apparatus in our sensitivity measurements of the SPECsfs NFS benchmark. A guiding engineering principle used in building the apparatus was to re-cycle as much of the user-level GAM code and LCP as possible. The alternative approach of adding delays and calibrating Myricom's device drivers and LCP was rejected as too time consuming to complete in the context of this thesis.

Our approach to building the apparatus was to insert the GAM layer inside the Solaris kernel. We created a kernel module which contained the user-level GAM code, slightly modified to run in the kernel, and then layered the STREAMS TCP/IP on top of it. However, modifications had to be made to the semantics of the GAM layer in order to accommodate placing active messages in the kernel.

The user level GAM layer required three semantic changes in order to most easily accommodate the STREAMS abstraction while still providing controllable delays. First, the request-reply model was removed from the code. The elimination of request-reply caused the second change, the removal of reliability semantics. Finally, the buffering semantics for medium messages required an extra copy on the receive side.

## Construction

Figure 2.6 shows the architecture of the TCP/IP-GAM apparatus. A number of STREAMS and character Solaris kernel drivers are required. Two drivers from Myricom are required to boot and control the LANai card (not shown). The Active Message driver implements most of the GAM functions. In order to more easily handle control operations, the Active Message driver is a simple character driver. STREAMS drivers require special messages in order to send control information; these are clumsy to use. However, as a character driver, the Active Message driver is unable to interface to the STREAMS subsystem directly. Therefore, a pseudo-Ethernet STREAMS driver was constructed to interface to the IP layer. The Ethernet driver was modeled on the Lance Ethernet driver provided in the Solaris source code. The Lance uses some fast-paths not provided in the normal sample drivers. The interface between the Ethernet and Active Message drivers is a modification of the GAM functions.

The Unix STREAMS model assumes a number of modules which are connected by queues. Figure 2.6 shows the relationship between these modules. The STREAMS framework contains the notion of layering. Each module has a "down" direction, towards the device, called the *write side*. The inverse is the "up" direction, called the em read side, which moves data toward the user process.

In addition to the two directions, there are two types of reads and writes: put procedures and service procedures. The main difference between put and service procedures is in the scheduling

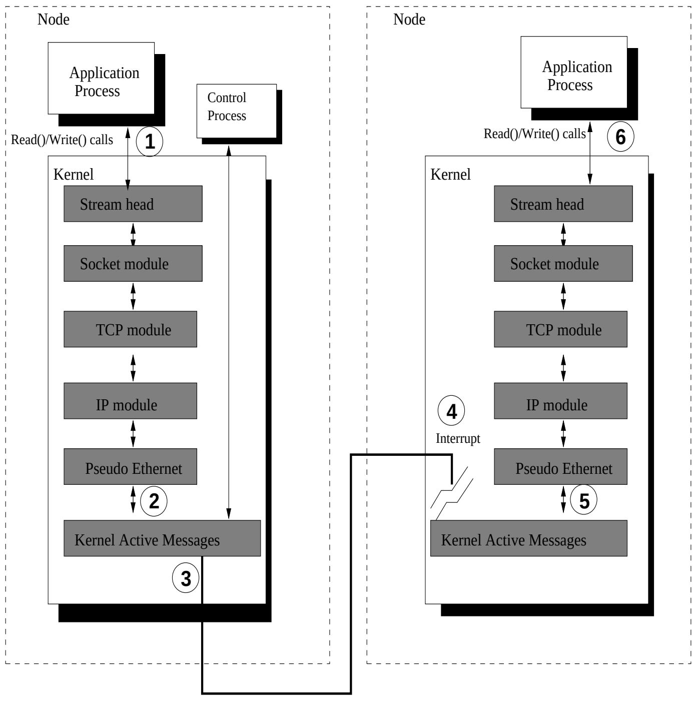


Figure 2.6: **TCP/IP Apparatus Architecture**
*This figure shows the software architecture of the TCP/IP emulation environment. Two Solaris kernel modules are used. One simulates and Ethernet driver and the second runs Kernel-to-Kernel Active Messages. The figure shows the path needed to send a message. After the application passes the message to the kernel(1), it eventually ends up at the pseudo-ethernet driver (2) which calls the kernel active message driver (3). After crossing the Myrinet, the receiving LANai interrupts the host (4), which invokes the poll routine of the kernel active message driver. The driver then passes it through the STREAMS sub-system (5) and eventually the message ends up at the receiving application (6).*

of the operation. The put procedure is called directly by the preceding module, while the service procedure is called via the STREAMS scheduler.

Tracing the path of a `write` system call in Figure 2.6, after the write call at (1), the Socket layer calls the write-put procedure of the TCP module, which calls the write-put procedure of the IP module, which calls the write-put procedure of the pseudo-Ethernet driver. Finally, at (2), the pseudo-Ethernet driver calls `am_request` with the IP packet as the data for the medium active message. In the normal case, the service procedures are not called. The kernel Active Message module copies the message into the LANai firmware queue at (3). The current implementation thus requires 2 copies on the send side.

When the receiving LANai sees a message in the receive queue, it generates an interrupt. The kernel vectors control to the Active Message poll function, `am_poll`, at (4). `Am_poll` in turn calls a handler in the pseudo-Ethernet driver, at (5). At this point, the pseudo-Ethernet driver allocates a STREAMS message structure and copies the packet into it. Next, the driver calls the read-put procedure of the IP module, and the messages eventually reaching the stream head. So far, all the receive processing has taken place inside the interrupt handler. Finally, when the application calls the `read` call, the message is copied into the application's memory.

Encapsulation is relatively straightforward. A Maximum Transmission Unit (MTU) of 3584 bytes (3.5K) was chosen because this represents the smallest unit that achieves the maximum bandwidth. We shall explore bandwidth in the calibration section.

A number of problems occurred when layering the STREAMS framework on top of the Active Message abstraction. The basic problem stems from interfacing the STREAMS notion of queues on top of the request-reply model. Modifications to GAM include eliminating the request-reply model and reliability guarantees. In addition, the GAM longevity model for medium messages did not support the STREAMS abstraction well, therefore the pseudo-Ethernet driver makes copies of the messages. The next paragraphs describe the request-reply semantics, reliability, longevity problems and the workarounds.

**Request-Reply Semantics** The basic problem with maintaining request-reply semantics is that the matching of requests to replies is difficult because the STREAMS subsystem can send data in an interrupt handler. The following example illustrates this problem. An Internet Control Message Protocol (ICMP) ping request arrives. In the interrupt handler, the IP module calls the ICMP routine. Still in the interrupt handler, the ICMP routine responds to the ping request by calling the write-put routine of the pseudo-Ethernet. All this happens before control is returned to the Active Message module. In the STREAMS model, the higher-layers can send an arbitrary number of messages in response to an incoming message, thus violating a semantic tenet of the GAM abstraction: that no requests can be sent during the execution of a reply handler.

While there are different possible solutions to the problem, the easiest to implement and most compatible with the STREAMS framework was to simply eliminate the request-reply model. All messages become “requests”, and are deposited directly into the send queue of the LCP. The notion of a “reply” message was eliminated from the code running on the host.

**Reliability** Recall that reliability was built in the Active Message layer by a combination of request-reply semantics linked to storage management. Once the request-reply model was discarded, maintaining reliability was dropped from the system, primarily because the increase in storage management complexities was not seen as worth the engineering effort required. Higher layers, such as TCP and RPC, must be used to provide reliability semantics in the TCP/IP-GAM system.

In addition to the difficulties of storage management, blocking was also a problem with incorporating the user-level GAM implementation into the kernel. The reliability of GAM can require blocking the current thread for extended periods. If the number of outstanding requests is too high, GAM spin-waits until an incoming reply signals a request buffer is “free”. Although this blocking is fine, even beneficial, for user-level programs, spin-waiting a thread for an arbitrary time is not permitted inside the kernel—such behavior can crash the entire system. The kernel GAM layer was thus modified to simply discard messages when outbound LCP queue is full. Although the STREAMS system provides for a “queue-full” signal to be propagated to higher modules, the added complexity of using this mechanism was judged to be not worth the engineering effort.

**Data Longevity** Data longevity semantics in the GAM abstraction effectively placed enough restrictions that copies had to be made on both the send and receive sides. On the sending side, the GAM model returns control to the sender once (1) the data area can be re-used for data storage and (2) the system has enough buffering to accept the message. Although the previous changes to the GAM layer eliminated these restrictions, the decision was made that the extra performance gained from zero-copy (inside the kernel) sends was an excessive engineering effort needed for this apparatus.

A second issue arises with attempting zero-copy sends on the Sun UltraSPARC workstation (the sun4u architecture). In a sun4u environment, I/O devices have their own address space; they cannot access memory in an arbitrary fashion. Instead of mapping and unmapping STREAMS buffers on every send, the AM driver copies the data into a fixed, I/O addressable region. Given the very fast (180MB/s) rate of the UltraSPARC memory system, this copy is not too expensive (20 $\mu$s for a 3.5 KB fragment).

On receives, the medium message data area is only valid for the life of the handler. This clashes with the STREAMS notion that a STREAMS message buffer exists in a single address space of the kernel independent of any particular queue. Thus, in order to implement zero-copy receives, the active message layer code would require modification to not free buffers upon return of control from the handler. As with the send case, the effort needed to construct a zero-copy apparatus was deemed not worth the added development time and risk.

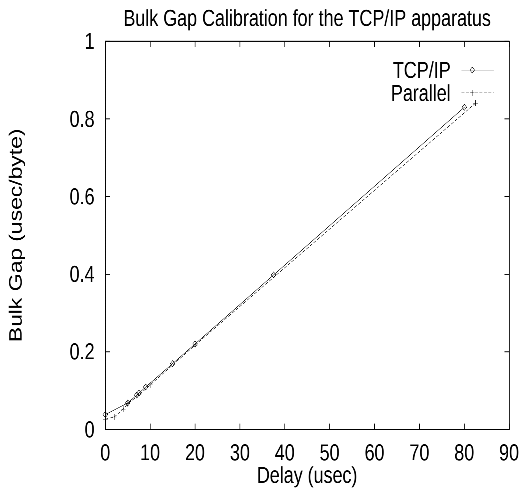


Figure 2.7: **Calibration of bulk Gap for TCP/IP-GAM apparatus**
*This figure shows the empirical calibration for bulk Gap for the TCP/IP apparatus. The dependent variable shows the added delay in $\mu$s per 100 bytes of packet size. The independent variable is the Gap expressed in $\mu$s per byte (1/bandwidth). The figure shows the TCP/IP-GAM apparatus for adjusting bulk Gap is quite accurate. The basic parallel program apparatus calibration is shown as well, demonstrating that the two systems are near equivalent as to Gap adjustment.*

**Calibration**

Because the LCP of the TCP/IP-GAM apparatus is taken from the parallel programming apparatus, we know that $L$ and a network-limited $g$ are identical, so we do not measure those parameters again. However, unlike the parallel programming apparatus, $o$ is substantially different. As can be seen from Figure 2.6, there are many software components involved in sending a message; the result is a large software overhead.

Since we know $L$, we can compute $o$ from a simple round trip time. Measurements show a mean RTT of 340 $\mu$s. With $L$ at 5 $\mu$s we can deduce $o$ as roughly 82 $\mu$s. Unlike in user-space, calibration of the delay loop inside the kernel can be tricky because many floating-point math routines are not supported in the kernel. Fortunately, the Solaris Device Driver Interface (DDI) contains a time-calibrated spin loop, `drv_usecwait`. It was intended for short waits to slow devices, but serves as a calibrated spin-loop quite well.

We must re-calibrate $G$, because the increases in $o$ may affect the range for which changes to the LCP bulk data handling loop affect $G$. We use the same methodology as the parallel apparatus. We cannot control the fragment sizes used by the kernel, however. Our gap experiment thus sends a single large block of data (10 MB), with each write call sending 8KB at a time. The observed time to report the entire transfer on the sender is taken as the delivered bandwidth. Figure 2.7 shows the results of the experiment. The parallel program Gap calibration is also plotted for comparison. Notice that for much of the Gap range, the two lines are nearly identical, showing that even with the different kernel fragmentation algorithm the Gap apparatus is quite accurate.

## 2.4 Factor Design

Once we have a calibrated, tunable apparatus, we face the question of how to vary each of the LogGP parameters. The space of possible settings is quite large. If we ignore $P$, there are still four parameters, each of which can be scaled by an order of magnitude. For example, we can vary $o$ from 5-100 $\mu$s and still obtain meaningful results.

In the terminology of experiment design, a *factor* is a variable that affects the outcome of the experiment. In this thesis, each of the networking components of the LogGP parameters, $L$,$o$,$g$,$G$, is clearly a factor. In addition, another dimension, or factor, we can control is the program used as a benchmark, e.g., Radix, EM3D, NowSort. Factors can take on different *levels*. For example, $o$ could take on levels of 5, 10, and 25 $\mu$s. Along the benchmark dimension, the "levels" would be the programs used as benchmarks. There are number of ways to explore the entire space covered by factors and levels. We provide a brief background here.

The traditional ways of combining factors and levels results in three classes of experiments: *simple*, *full*, and *fractional* [57]. In all types, all the factors are varied. The real difference between the methods is how the levels are adjusted. In the simple design, we first assign each factor a constant level. Next, we vary each factor by a large number of levels, while keeping the other factors at their constant level. *Simple experiments* have the advantage that we can observe the effect of each factor is isolation, but has the disadvantage that it may miss interactions between factors, i.e., factors may not be independent. For example, in the LogGP case, we know that if we inflate $o > g$, then $g$ is no longer observable as a separate parameter; it degenerates into $o$. Therefore, the LogGP model tells us that sensitivity experiments where we simultaneously inflate both $o$ and $g$ such that $o > g$ will probably not yield different results than inflating $o$ alone.

Another class of experimental design is called the *full factor method*. In this type of design, all factors are varied by all levels. Clearly, this experiment design has the disadvantage of requiring the most experiments. It does, however, have the advantage of capturing all interactions between the factors. For example, in our experimental designs space, we have five factors (the LogGP parameters and benchmark), each of which can take on, say 8 levels. A full-factor design would require running

$8^5$ or 32768 experiments. In our case, such a design clearly requires too many experiments.

The final class of designs are called *fractional designs*. In these types of experiments, the number of levels is reduced from the full factor design while still simultaneously varying factors and levels. The minimalist case is called a $2^k$ design. If we have $k$ factors and we vary each factor by two levels, then the number of experiments performed is $2^k$. In our case such a design would require only 32 experiments; quite a small number. However, such an experiment design leaves many unanswered questions. For example, we cannot see knees or plateaus in the data because by definition we are only sampling at 2 levels for each factor. Also, in our case, we would only sample 2 benchmarks! The classic $2^k$ design, as well as many other fraction designs, are best suited when the interactions between factors is largely unknown and the purpose of the experiments is thus an exploration of the interactions between parameters.

This thesis uses a simple design, as opposed to a fractional design. There are two advantages of this design. First, we can observe a wide variety of applications by fully scaling the benchmark axis. This is the most crucial axis to scale, as we do not claim to have a representative workload of all applications. By observing the reaction of a wide range of programs, we can better classify new programs as similar or dissimilar to our existing set. In addition, our wide range of benchmarks allows us to quantify the sensitivity of a “worst-case” program for each networking parameter. For example, EM3D(read) is a program that performs only blocking reads. It should therefore be a program that is “worst-case” with respect to sensitivity to $L$.

A second advantage of a simple experiment design is that it allows us to explore more factor levels at the edge of our apparatuses’ operational limits. For example, with parallel programs we are quite interested at the response of programs near the lowest overhead limit of our apparatus. If we see an insensitive region, we may concluded that a new class of low overhead protocols has solved the “overhead problem”, or at least some other system component is the bottleneck. On the other hand, if we still observed a sensitive region in the low overhead regime, we may conclude instead that further reductions in overhead are warranted. With a fractional design, we are more likely to miss these regions. Likewise, in the NFS context we are interested in performance near peak operations per second. The shape of the response-time vs. throughput curve near the peak operations per second for different $o$ is most interesting; it tells us how useful lowering $o$ is in performing graceful load degradation.

The main disadvantage of our simple design is that we may miss interactions between the factors. However, the LogGP model can compensate us somewhat, in that it gives us a way to reason about what the interactions should be. Although we have chosen not to explore these interactions, it would be useful as a validation of the LogGP model.

The most important class of factor-level combinations missing from this work are that of real machines. For example, we could easily emulate a machine running a TCP/IP stack on 100 Mb Ethernet. This class of machines will be quite important in the future, as it represents a very cost-effective point in the design space. Likewise, our apparatus could emulate a 622Mb ATM cluster of workstations. Running experiments on these common designs would not only provide additional validations of our experiments, it would allow a near direct comparison of the cost-performance of these technologies (e.g. ATM vs. Ethernet). However, although such point-comparisons are quite useful, many other studies already have performed such comparisons [1, 49, 61]. For the purposes of this thesis, we view the uniqueness of isolating the impact of each parameter as more important that making point-wise comparisons of different technologies.

## 2.5 Other Models

The LogGP model is not the only model available to describe program behavior. In this section, we explore other models. We first describe the models, as well as relate their purposes, strengths and weaknesses in the context of this thesis. We also describe why this thesis uses the LogGP model.

Most of the models presented here were developed in the context of parallel program design, must be considered in light of that purpose. Previous to the invention of the models presented here, many parallel programs were written using the Parallel Random Access Machine (PRAM) model. The problem with the PRAM model is that it ignores real machine characteristics and thus encourages algorithm designers to exploit characteristics of the PRAM machine that result in poor performance on real machines.

Although a reaction the unrealistic assumptions of the PRAM model, many of the models explored in this section do not have parameters that correspond well to a variety of real machine components. Out of a fear that additional parameters will make the models cumbersome to use, many of the models attempt to capture the limitations of real machines in only 1 or 2 parameters. The purpose of the parameters is to encourage algorithm designers to creating algorithms that run well on real machines; the parameters are not intended to model the machines per say. The lack of architectural focus is not a fault of the models, but limits their applicability for understanding the role of machine architecture on application performance.

### 2.5.1 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model [101] attempts to bridge theory and practice with an very simple model. Computation is divided into a number of "supersteps" that are separated by global barriers. Each superstep must be at least $L$ units long, and messages from one step cannot be used until the next step. Like the LogGP model, BSP does not model any specific topology. In addition to the restriction that a superstep must last at least $L$ time units, messages can only be sent or received at an interval of $g$ during a given step.

Although proven useful for algorithm design, the BSP's restricted architectural focus would greatly limit this thesis. The architectural parameters are only $L$ and $g$. In a real programs $o$ and $G$ are often limitations that would be missed by the BSP model.

In addition, it would be difficult to categorize all parallel programs into the BSP framework. Most challenging are the supersteps. For example, several of the applications in this thesis use task-queue models that do not fit well into the BSP framework. The task-queue programs have only a few long phases where communication is overlapped with computation; in some programs there is only a single phase [97]! Thus, although a program written in the BSP framework will make efficient use of machine resources, the model will not reveal much about application behavior or architectural characteristics.

### 2.5.2 Queue Shared Memory

Like BSP, the Queue Shared Memory (QSM) model is also designed for parallel programming [46]. This model extends the PRAM model to include queuing effects at shared memory locations. The model divides memory into local and global portions. Operations on the global memory take unit time, unless they operate on the same location, in which case they are queued and serviced at a rate of $g$. Thus, the only two parameters in this model are $g$ and $P$, and the only network parameter is $g$.

A major flaw from an architectural perspective with this model is that the $g$ parameter is on a per-word basis. The QSM developers correctly note that a machine with many memory banks per processor can adequately approximate such a model, and use the Cray vector supercomputers as an examples. However, as technology trends move more memory capacity per chip [50], it will become increasing difficult at an architectural level to sustain many memory banks. This flaw in the QSM model could be easily corrected by modifying the model to a point where $g$ is sustained for every global memory operation, which would encourage algorithm designers to avoid remote memory as much as possible.

### 2.5.3 LoPC and LoGPC

The LoPC and LoGPC models [43, 78] extend the basic LogP model with contention parameters. The original paper, [43] added the $C$ parameter to model contention effects at the endpoints. In large parallel machines with all-to-all patterns, these contention effects can result communication times that are 50% greater than predicted by a straightforward LogGP model [39]. Note that these contention effects get worse with increasing overhead.

The LoGPC model extends the contention effects into the network. However, in order to model these effects, the work makes certain assumptions about the network. Although the model captures a large class of networks by abstracting k-ary d-cubes, other topologies, such as certain expander networks, are not modeled. Still, for low-order dimension-routed networks, network congestion can be a serious limitation.

Of all the models presented here, LoPC is perhaps the most interesting for the purposes of this thesis. Contention effects can become a significant fraction of the time on large machines. This observation is not new [84], but the recent work on LoPC and LoGPC better quantify this effect.

However, as was shown in [39], as well as in the LoPC model itself, for the range of machines of interest to this study, (16-32 nodes) the endpoint contention effects are minimal. For example, the predicted vs. actual communication cost of the radix sort program on a 32 node CM-5 differed by only 9%. Only at larger machines sizes (256 and 512 nodes) do contention effects dominate, resulting in predictions that are up to 50% inaccurate. However, the standard LogGP model is quite suitable for this study because of the machine sizes used.

### 2.5.4 Queuing Theory

In the world of LAN and WAN networks, queuing theoretic approaches dominate [57, 63, 65]. The problem with using this class of models for parallel programs is that the basic assumptions about program behavior do not mesh well with what many parallel programs do. For example, queuing theoretic models assume that the output of a program can be modeled by some stochastic process. That is, network data is generated as some random function of time. Often, poisson processes are used because the mathematics remain tractable while constructing "realistic" models. In addition, much of queuing theory is built on the assumption that the system is in steady-state operation. This allows the modeler to close the resulting system of equations. The nature of many parallel programs, however, is often not captured well by stochastic processes. For example, bulk-synchronous parallel programs alternate between communication and computation phases; thus all communication occurs in large bursts. Communication events are not random, nor is there a global steady-state. We shall examine application behavior in detail when we examine the sensitivity results.

However, we do use a queuing theoretic model in the study of NFS servers. In this case, the benchmark itself is influenced by the stochastic model of program behavior. This self-referential assumptions are part of the reason why the queuing model works well for the benchmark. We will examine this phenomenon in Section 5.6.

## 2.6 Related Methodologies

The section describes the methodologies of related work. Although impossible to cover all related research, we highlight four studies that have results most relevant to our work. We show that although both the application and design spaces are enormous, nearly all of the studies can be placed along the two axis of experiment design outlined in Section 1.1.

Where possible, we explain the results of these studies using the LogGP model. Although most of the studies did not use LogGP, we show that most of the results can be interpreted in a LogGP framework. Casting other results into a LogGP framework also serves as additional validation of the model.

### 2.6.1 Holt

The focus of [52] is quite close in spirit to this thesis. The main questions in that work were how shared memory parallel programs would respond to different abstract machine parameters. The study was application-centric and used simulation as the evaluation method. Its abstract parameters are very close to those of the LogGP model. Because the programs were written using shared memory, and the machines studied were cc-NUMA designs, the study introduced a new term, occupancy. In terms of the LogGP model, occupancy of a machine's cache controller lies somewhere between overhead and gap. Occupancy can limit the peak message rate. Even with speculative and out-of-order processors, a high occupancy can also stall the processor, causing an increase in overhead. The $L$ of the Holt model was unchanged that in LogGP.

In addition to a similar network model, Holt's experiment paradigm was quite similar to this thesis. Instead of "slowdown", however, parallel efficiency was used as the application-centric metric. Even without the initial run-time, slowdown can be derived from this metric, as they both share the same denominator. Parallel efficiency, however, has a number of problems when used as a metric compared with slowdown. The primary reason is that it obscures the real question, which is machine sensitivity to the parameters.

The Holt study found that occupancy, as opposed to latency, was the dominate term affecting parallel program performance. Much like software overhead, occupancy in shared-memory programs is difficult to tolerate [32]. It found that very high $L$, into the 1000's of machine cycles, would reduce efficiency by 50%, a factor of 2 in slowdown. However, a much smaller increase in controller occupancy, into the 100's of cycles, could reduce efficiency by up to 50% as well.

The Holt study also developed a number of analytic models to investigate the effects of increased latency and occupancy. The simplest models used a simple frequency-cost pair scheme and ignored contention effects. On a 64 processor machine, this model was off by up to 40%. A more accurate queuing model reduced the difference between the simulation and the model to less that 15%. Unfortunately, the model was only used for one simple application, so we can not conclude about the general accuracy of the model compared to the simulation.

### 2.6.2 Chang

A recent work [21] examined NFS performance over congested ATM networks. The goal of this work was to determine the effect of various ATM credit based flow control schemes on NFS. The work was an application-centric simulation study. A trace-fed simulation from various NFS tasks (e.g. a compile) was used as the evaluation. The study used run-time of the entire task as the metric for evaluation, as opposed measuring individual operations of the NFS protocol. The most related part of the methodology was that the study examined the impact of scaling an abstract parameter, $L$, in addition to point-wise comparisons of flow control schemes. The study did not explore other parameters, however.

The study found that a high $L$, over 10 milliseconds, was detrimental. However, low $L$, in the $\mu$s range, was not found to impact performance. In addition, the study found that a combination of TCP backoff algorithm and segment sizes can slow performance down by as much as 30%. Because of the custom workloads, however, making a direct comparison of absolute sensitivities to our NFS results is difficult.

### 2.6.3 Ahn

This study [1] compared two TCP congestion control and avoidance strategies, TCP-Reno and TCP-vegas. The study examined the effects of these different congestion avoidance strategies on FTP traffic. Much like this thesis, a slowdown layer, called the “hitbox”, was interposed under the TCP/IP stack to emulate WAN links. The work was application-centric and used an emulation methodology. The hitbox was built as an interposition layer between the IP layer and the device driver in the BSD operating system. Unlike this thesis, however, the independent variable of the experiment was not a set of abstract parameters. Rather, the independent variable was the TCP algorithm.

The hitbox can abstract the link bandwidth, propagation delay, and bit error rate. The methodology to construct the emulator was different from this thesis in that many links were used to construct the network. That is, each link was designed to emulate a single wide area link, and many hosts with multiple links were used to emulate a WAN. This approach is contrasts our approach where we emulate the entire $L$ using a single delay.

The metric used was rather simple, the time to FTP a 512KB file. In other experiments, the average of many simultaneous transfers was used as the dependent variable. Although the dependent variable in the experiment was application-centric, the study attempted to answer several network-centric questions as well. These included which TCP algorithm transmitted more bytes through the network, as well as which resulted in longer queues at the switches.

The study found that TCP Vegas can increase delivered bandwidth by 3-5% over the Reno version. The additional overhead of Vegas over Reno was described but not measured. In addition, the study found that Vegas resulted in an easier load on the network switches, in terms of offered bandwidth, than Reno.

In spite of the excellent apparatus, the study was somewhat disappointing because the application-centric focus was not fully investigated. Only measured competing FTP traffic was the subject of the study. However, the hitbox emulation system could have measured the impact of different network designs and algorithms on a wide variety HTTP, NFS and multimedia traffic as well.

The construction of the hitbox raised the issue of the apparatus changing what it’s trying to measure. Because they did not use a separate network processor to emulate network parameters, the hitbox itself added communication overhead. The study concluded that the additional overhead was only 3%, but the calibration methodology was slightly dubious. The study measured the slowdown of a quicksort while the background idle hitbox was running. A better methodology would have been a direct measurement of the change in $o$ as a function of the hitbox operations.

### 2.6.4 Hall

The impact of specific networking technologies (ATM, Autonet, FDDI) on NFS was examined in [49]. This study used an application-centric, direct measurement methodology. They used two workloads. The first was NFSstone [93], the precursor to SPECsfs. Some of their most interesting data, however, came from the direct measurements of a 200 workstation production system serving about 150 people.

The methodology of measurements of a production system differ sharply from majority of previous work, and in some sense is also the most representative. For example, if real systems do not behave like poisson processes, this will be captured in production system. The study found that the observed RTT in productions systems had a much larger variance in comparison to their controlled experiments.

Their conclusions are quite similar to ours: CPU overhead is a dominant factor in NFS performance. One of the targets of the study was the effects of overhead reduction by increasing the Maximum Transmission Unit (MTU) size. The study found that reducing overhead by increasing MTU size can greatly impact performance. However, the data shows the relationship is non-linear. Increasing the MTU from the 1.5 KB Ethernet MTU to the 4.3 KB FDDI MTU resulted in a large improvement in the average response time for a mix of NFS operations. However, the improvement from a 4.3 KB MTU to the 9 KB ATM MTU was much less, and in some cases actually degraded performance. Unfortunately, they did not present the operation mix for their live experiments.

It is important to note that interpretation of the results in the original work was not cast in term of overhead. Rather, the work discussed “end-system” performance:

> ... in order to achieve maximum benefit from modern and fast networks in the future a big effort must be addressed towards the improvement of the end systems.

This quote highlights the importance of simple models, such as LogGP, in understanding machine performance. Without a simple model, characterizing complex systems in meaningful ways is difficult task. For example, the Hall study provided detailed breakdowns of different pieces of the NFS operations, but did not lump them into the categories of overhead and latency. Although easy to discern the difference from their data, such simple categorization would have made explaining their results much more intuitive.

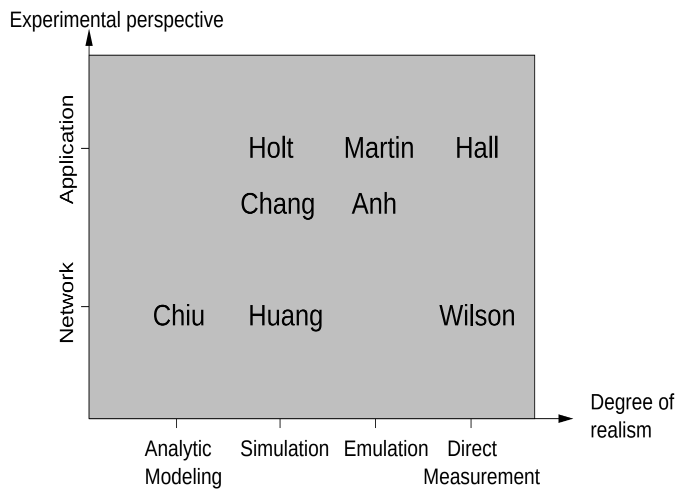


Figure 2.8: **Methodologies in Context**
*This figure shows were various works fit in the space of network experiment methodologies. The x-axis orders methodologies by degrees of realism, while the y-axis shows the focus of the dependent variable.*

### 2.6.5 Summary

In this section, we return to the network experiment conceptual framework developed in Section 1.1. Figure 2.8 shows a number of studies placed in our framework of experiment design. The experiments are: Holt [52], Martin [73], Hall [49], Chang [21], Ahn [1], Chiu [24], Huang [54] and Wilson [62, 66, 67]. When placing experiments in this space, we have used the primary focus of the study to determine the axis. Although some studies [1, 52, 73] used multiple techniques at once, it is interesting that even these have one axis clearly dominate the other.

We can see that experiments cover nearly the full spectrum of experiment design. A point relevant to this thesis is that there are few experiments which categorize application-centric sensitivities to abstract parameters. Thus, there is little systematic exploration of the system design space. Instead, most studies compare single instances of different systems or algorithms, e.g. TCP Vegas vs. Reno [1], credit vs. rate based flow control [21], or congestion avoidance algorithms [24]. A point which does not need elaboration is that, of course, many research groups have an axe to grind with regard to these systems.

There is a notable lack of application-centric experiments which rely on analytic modeling. Although [24], and to a lesser extent [56] model the effects on applications somewhat, the actual concern is not applications per say, but the collective effects of many applications on the network infrastructure. Speculating a bit, the author believes this is because applications are difficult to model. Indeed, only recently has the network community accepted the work in [66]; that poisson processes are an inaccurate model for aggregating application traffic. It is clear from our examination of the literature that application behavior has not received the same attention in the network or parallel program communities as in the architecture or database communities. The adoption of well defined benchmark suites would go a long way towards solving this problem.

# Chapter 3

# Split-C/AM Program Sensitivity

*An experimental science is supposed to do experiments that find generalities. It's not just supposed to tally up a long list of individual cases and their unique life histories. That's butterfly collecting.* —Richard C. Lewontin, The Chronicle of Higher Education, February 14, 1997

This chapter contains an examination of the sensitivities of a suite of parallel programs written in the Split-C language or programmed in Active Messages directly. We first characterize the programs in terms of Split-C operations and parallel program orchestration techniques. We then describe the programs in detail, followed by some simple analytic models of them. Next, we present the results of our sensitivity experiments and comment on the accuracy of the analytic models. Finally, we summarize the results of this chapter.

## 3.1 Characterization

With a methodology in place for varying communication characteristics, we now characterize the architectural requirements of the Split-C application suite. Split-C is a parallel extension of the C programming language that provides a global address space on distributed memory machines. Split-C (version 961015) is based on GCC (version 2.6.3) and Generic Active Messages (version 961015), which is the base communication layer throughout. Note that two of the programs use Active Messages directly, bypassing the Split-C layer. Although the programming model does not provide automatic replication with cache coherence, a number of the applications perform application-specific software caching. The language has been ported to many platforms [2, 72, 103, 104]. The sources for the applications, compiler, and communication layer can be obtained from a publicly

| **Program** | **Description** | **Input Set** | **16 node Time (sec)** | **32 node Time (sec)** |
|---|---|---|---|---|
| Radix | Integer radix sort | 16 Million 32-bit keys | 13.66 | 7.76 |
| EM3D(write) | Electro-magnetic wave propagation | 80000 Nodes, 40% remote, degree 20, 100 steps | 88.59 | 37.98 |
| EM3D(read) | Electro-magnetic wave propagation | 80000 Nodes, 40% remote, degree 20, 100 steps | 230.0 | 114.0 |
| Sample | Integer sample sort | 32 Million 32-bit keys | 24.65 | 13.23 |
| Barnes | Hierarchical N-Body simulation | 1 Million Bodies | 77.89 | 43.24 |
| P-Ray | Ray Tracer | 1 Million pixel image 16390 objects | 23.47 | 17.91 |
| Mur$\varphi$ | Protocol Verification | SCI protocol, 2 procs, 1 line, 1 memory each | 67.68 | 35.33 |
| Connect | Connected Components | 4 Million nodes 2-D mesh, 30% connected | 2.29 | 1.17 |
| NOW-sort | Disk-to-Disk Sort | 32 Million 100-byte records | 127.2 | 56.87 |
| Radb | Bulk version of Radix sort | 16 Million 32-bit keys | 6.96 | 3.73 |

Table 3.1: **Split-C Applications and Data Sets**
*This table describes our applications, the input set, the application's communication pattern, and the base run time on 16 and 32 nodes. The 16 and 32 node run times show that most of the applications are quite scalable between these two machine sizes.*

available site [1].

To ensure that the data is not overly influenced by startup characteristics, the applications must use reasonably large data sets. Given the experimental space we wish to explore, it is not practical to choose data sets taking hours to complete; however, an effort was made to choose realistic data sets for each of the applications. We used the following criteria to characterize applications in our benchmark suite and to ensure that the applications demonstrate a wide range of architectural requirements:

- **Message Frequency:** The more communication intensive the application, the more we would expect its performance to be affected by the machine's communication performance. For applications that use short messages, the most important factor is the message frequency, or equivalently the average interval between messages. However, the behavior may be influenced by

[1] `ftp.cs.berkeley.edu/pub/CASTLE/Split-C/release/sc961015`

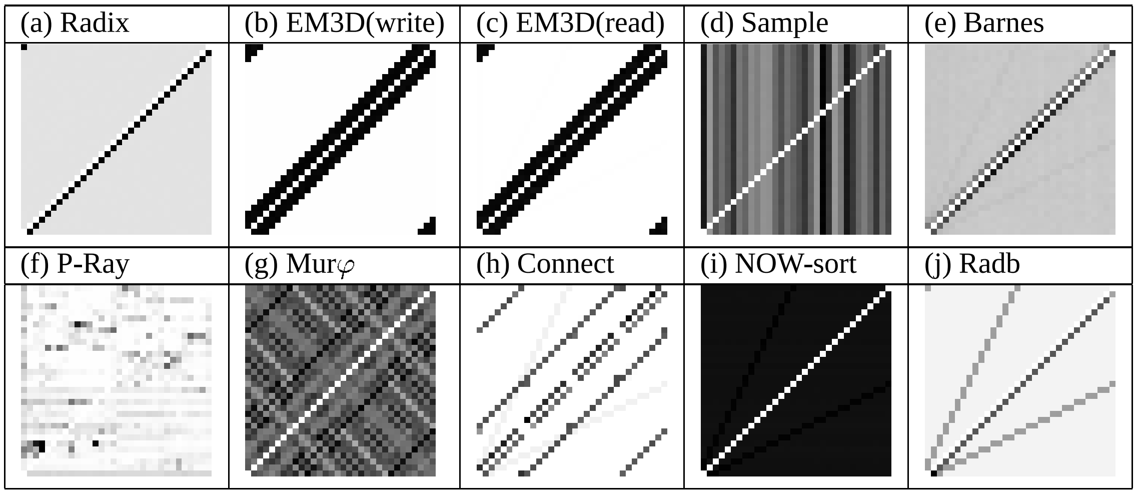


Figure 3.1: **Split-C Communication Balance**
*This figure demonstrates the communication balance between each of the 32 processors for our 10 Split-C applications. The greyscale for each pixel represents a message count. Each application is individually scaled from white, representing zero messages, to black, representing the maximum message count per processor as shown in Table 3.2. The $y$-coordinate tracks the message sender and the $x$-coordinate tracks the receiver.*

the burstiness of communication and the balance in traffic between processors.

- **Write or Read Based:** Applications that read remote data and wait for the result are more likely to be sensitive to latency than applications that mostly write remote data. The latter are likely to be more sensitive to bandwidth. However, dependences that cause waiting can appear in applications in many forms.

- **Short or Long Messages:** The Active Message layer used for this study provides two types of messages, short packets and bulk transfers. Applications that use bulk messages may have high data bandwidth requirements, even though message initiations are infrequent.

- **Synchronization**: Applications can be bulk synchronous or task queue based. Tightly synchronized applications are likely to be dependent on network round trip times, and so may be very sensitive to latency. Task queue applications may tolerate latency, but may be sensitive to overhead. A task queue based application attempts to overlap message operations with local computation from a task queue. An increase in overhead decreases the available overlap between the communication and local computation.

- **Communication Balance**: Balance is simply the ratio of the maximum number of messages

| Program | Avg. Msg./ Proc | Max Msg./ Proc | Msg./ Proc/ ms | Msg. Interval ($\mu$s ) | Barrier Interval (ms) | Percent Bulk | Percent Reads | Bulk Msg. (KB/s) | Small Msg. (KB/s) |
|---|---|---|---|---|---|---|---|---|---|
| Radix | 1,278,399 | 1,279,018 | 164.76 | 6.1 | 408 | 0.01% | 0.00% | 26.7 | 4,612.9 |
| EM3D(write) | 4,737,955 | 4,765,319 | 124.76 | 8.0 | 122 | 0.00% | 0.00% | 0.6 | 3,493.2 |
| EM3D(read) | 8,253,885 | 8,316,063 | 72.39 | 13.8 | 369 | 0.00% | 97.07% | 0.0 | 2,026.9 |
| Sample | 1,015,894 | 1,294,967 | 76.76 | 13.0 | 1,203 | 0.00% | 0.00% | 0.0 | 2,149.2 |
| Barnes | 819,067 | 852,564 | 18.94 | 52.8 | 279 | 23.25% | 20.57% | 110.4 | 407.1 |
| P-Ray | 114,682 | 278,556 | 6.40 | 156.2 | 1,120 | 47.85% | 96.49% | 358.5 | 93.5 |
| Connect | 6,399 | 6,724 | 5.45 | 183.5 | 47 | 0.06% | 67.42% | 0.0 | 152.5 |
| Mur$\varphi$ | 166,161 | 168,657 | 4.70 | 212.6 | 11,778 | 49.99% | 0.00% | 3,876.6 | 65.8 |
| NOW-sort | 69,574 | 69,813 | 1.22 | 817.4 | 1,834 | 49.82% | 0.00% | 3,125.1 | 17.2 |
| Radb | 4,372 | 5,010 | 1.17 | 852.7 | 25 | 34.73% | 0.04% | 33.6 | 21.4 |

Table 3.2: **Split-C Communication Summary**
*For a 32 processor configuration, the table shows run times, average number of messages sent per processor, and the maximum number of messages sent by any processor. Also shown is the message frequency expressed in the average number of messages per processor per millisecond, the average message interval in microseconds,the average barrier interval in milliseconds, the percentage of the messages using the Active Message bulk transfer mechanism, the percentage of total messages which are read requests or replies, the average bandwidth per processor for bulk messages in kilobytes per second, and the average bandwidth per processor for small messages in kilobytes per second.*

sent per processor to the average number of messages sent per processor. It is difficult to predict the influence of network performance on applications with a relatively large communication imbalance since varying LogP parameters may exacerbate or may actually alleviate the imbalance.

### 3.1.1 Split-C Benchmark Suite

Table 3.1 summarizes the programs we chose for our benchmark suite as run on both a 16 and a 32 node cluster. Most applications are well parallelized when scaled from 16 to 32 processors. It is important to note the history of these applications when examining our results. All of the applications were designed for low overhead MPPs or NOWs. The program designers were often able to exploit the low-overhead aspect of these machine architectures in the program design. Each application is discussed briefly below.

- **Radix Sort:** sorts a large collection of 32-bit keys spread over the processors, and is thoroughly analyzed in [39]. It progresses as two iterations of three phases. First, each processor determines the local rank for one digit of its keys. Second, the global rank of each key is calculated from local histograms. Finally, each processor uses the global histogram to distribute the keys to the proper location. For our input set of one million keys per processor on 32 processors the application spends 98% of its time in the communication phases.

The communication density plot of Figure 3.1a is useful in understanding the communication behavior of this application. The darkness of cell $i, j$ indicates the fraction of messages sent from processor $i$ to processor $j$. The dark line off the diagonal reflects the global histogram phase, where the ranks are accumulated across processors in a kind of pipelined cyclic shift. The grey background is the global distribution phase. Overall, the communication is frequent, write-based and balanced.

- **EM3D:** EM3D [28] is the kernel of an application that models propagation of electromagnetic waves through objects in three dimensions. It first spreads an irregular bipartite graph over all processors. During each time-step, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and vice versa. We use two complementary versions of EM3D, one write-based and the other read-based. Both versions contain relatively short computation steps. The write-based EM3D uses pipelined writes to propagate updates by augmenting the graph with special boundary nodes. EM3D(write) represents a large class of bulk synchronous applications, alternating between local computation and global communication phases. The read version uses simple blocking reads to pull update information locally and does not need to create special boundary nodes. The locality of connectivity in the graph for both versions is indicated by the dark swath in Figures 3.1b and 3.1c.

- **Sample Sort:** is a probabilistic algorithm which sorts a large collection of 32-bit keys by first choosing $p-1$ "good" splitter values and broadcasting them to all processors. Every processor distributes its keys to the proper destination processor, based on the splitter values, and finally, a local radix sort is performed on the received keys. An interesting aspect of this application is the potential for unbalanced all-to-all communication as each processor potentially receives a different number of keys. This is reflected in the vertical bars in Figure 3.1d. For our input size, the local sort time is dominated by the distribution of keys to their proper destinations. For our input of 16 million keys Sample sort spends 85% of the time in the two communication phases.

- **Barnes:** Our implementation of this hierarchical N-Body force calculation is similar to the version in the SPLASH benchmark suite [110]. However, the main data structure, a spatial oct-tree, is replicated in software rather than hardware. Each timestep consists of two phases, a tree construction phase and an interaction phase among the simulated bodies. Updates of the oct-tree are synchronized through blocking locks. During the interaction phase, the processors cache oct-tree nodes owned by remote processors in a software managed cache. Communication is generally balanced, as the solid grey square shows in Figure 3.1e.

- **P-Ray:** This scene passing ray tracing program distributes a read-only spatial oct-tree over all processors. The processors evenly divide ownership of objects in the scene. When a processor needs access to an object stored on a remote processor, the object is cached locally in a fixed sized software-managed cache. Communication thus consists entirely of blocking read operations; the frequency of such operations is a function on the scene complexity and the software caching algorithm. The dark spots in Figure 3.1f indicate the presence of "hot" objects which are visible from multiple points in the scene.

- **Parallel Mur$\varphi$:** In this parallel version of a popular protocol verification tool [34, 97], the exponential space of all reachable protocol states is explored to catch protocol bugs. Each processor maintains a work queue of unexplored states. A hash function maps states to "owning" processors. When a new state is discovered, it is sent to the proper processor. On reception of a state description, a processor first checks if the state has been reached before. If the state is new, the processor adds it to the work queue to be validated against an assertion list.

- **Connected Components:** First, a graph is spread across all processors [69]. Each processor then performs a connected components on its local subgraph to collapse portions of its components into representative nodes. Next, the graph is globally adjusted to point remote edges (crossing processor boundaries) at the respective representative nodes. Finally, a global phase successively merges components between neighboring processors. The communication to computation ratio is determined by the size of the graph.

- **NOW-sort:** The version of NOW-sort used in this study sorts records from disk-to-disk in two passes [8]. The sort is highly tuned, setting a the MinuteSort world record in 1997. The sorting algorithm contains two phases. In the first phase, each processor reads the records from disk and sends them to the final destination processor. The perfectly balanced nature of the communication of phase 1 is shown by the solid black square in Figure 3.1i. The sort uses one-way Active Messages directly, sending bulk messages at the rate the records can be read from disk. Phase 2 of the algorithm consists of entirely local disk operations. Unlike the other applications, NOW-sort performs a large amount of I/O, so can overlap communication overhead with disk accesses.

- **Radb:** This version of the radix sort [2] was restructured to use bulk messages. After the global histogram phase, all keys are sent to their destination processor in one bulk message. Depending on network characteristics, use of these bulk messages can speed up the performance of the sort relative to the standard radix sort.

### 3.1.2 Characteristics

As summarized in Table 3.1, the applications represent a broad spectrum of problem domains and communication/computation characteristics. To quantify the differences among our target applications, we instrumented our communication layer to record baseline characteristics for each program (with unmodified LogGP parameters) on 32 nodes. Table 3.2 shows the average number of messages, maximum number of messages per node (as an indication of communication imbalance), the message frequency expressed in the average number of messages per processor per millisecond, the average message interval in microseconds, and the average interval between barriers as a measure of how often processors synchronize. Table 3.2 also shows the percentage of the messages using the Active Message bulk transfer mechanism, the percentage of the total messages which are a read request or reply, the average bandwidth per processor for bulk messages, and the average bandwidth per processor for small messages. Note that the reported bandwidth is for bytes transmitted through the communication layer as opposed to bandwidth delivered to the application, e.g., it includes headers.

Table 3.2 shows that the communication frequency of our applications varies by more than two orders of magnitude, and yet none of them are “embarrassingly parallel.” This disparity suggests that it is quite difficult to talk about typical communication behavior or sensitivity. Most of the applications have balanced communication overall, whereas others (Sample, P-Ray) have significant imbalances. Barnes and EM3D(write) are bulk synchronous applications employing barriers relatively frequently. Barnes, Mur$\varphi$, P-Ray, Radb and NOW-sort utilize bulk messages while the other applications send only short messages. Finally, EM3D(read), Barnes, P-Ray, and Connect do mostly reads, while the other applications are entirely write based. Applications doing reads are likely to be dependent on network round trip times, and thus sensitive to latency, while write based applications are more likely to be tolerant of network latency. Most of the applications demonstrate regular communication patterns. However, Connect and P-Ray are more irregular and contain a number of hot spots. While these applications do not constitute a workload, their architectural requirements vary across large classes of parallel applications.

## 3.2 Analytic Models

### 3.2.1 Overhead

To develop insight into our experimental results, we develop a simple analytical model of application sensitivity to added overhead. The model is based on the fact that added overhead is incurred each time a processor sends or receive a message. Thus, given an processor's base runtime, $r_{orig}$, the added overhead, $\Delta o$, and $m$, the number of communication events for each processor, we expect runtime, $r_{pred}$, to be:

$$r_{pred} = r_{orig} + 2m\Delta o$$

The factor of two arises because, for Split-C programs, all communication events are one of a request/response pair. For each request sent, the processor will incur an overhead penalty receiving the corresponding response in addition to the overhead for the sent request. If the processor is sending a response, it must have incurred an overhead penalty when it received the request message.

Given this model for the overhead sensitivity of individual processors, we extrapolate to predicting overall application runtime by making the following simplifying assumptions. First, applications run at the speed of the slowest processor, and second, the slowest processor is the processor that sends the most messages. Thus, by replacing $m$ in the equation with the maximum number of messages sent by a processor from Table 3.2, we derive a simple model for predicting application sensitivity to added overhead as a function of the maximum number of messages sent by any processor during execution.

The simple linear model presented above does not capture serial dependencies in the application. Our overhead model will thus tend to under predict run time due to *serialization effects*. For example, imagine a pipelined shift performed by all processors: processor zero sends to processor one, which waits for the message then forwards it to processor two, etc. The maximum number of messages sent by any processor is one, but the entire time of the operation is proportional to $P$. As we inflate overhead by $\Delta o$, the total time for this operation will increase by $P\Delta o$, not $\Delta o$ as predicted by the model. Serial dependencies such as the one outlined above are highly application specific and so we choose to use the a simple linear model. Section 3.3.1 quantifies the model's under prediction.

for all the applications.

### 3.2.2 gap

Developing a model for application sensitivity to gap presents more difficulties than developing the model for sensitivity to overhead. A processor is not affected unless it attempts to send messages more frequently than the gap. At this point, the processor must stall for a period waiting for the network interface to become available. Without more precise knowledge of inter-message intervals, the model for gap sensitivity depends on assumptions made about these intervals. At one extreme, the uniform model assumes that all messages are sent at the application's average message interval, $I$, from Table 3.2. In this case, the predicted runtime, $r^{(u)}_{pred}$, can be predicted as a function of total gap, $g$, the average message interval, $I$, the base runtime, $r_{base}$, and the maximum number of messages sent by any node, $m$:

$$r^{(u)}_{pred} = \begin{cases} r_{base} + m(g - I) & \text{if } g > I \\ r_{base} & \text{otherwise} \end{cases}$$

At the other extreme, the burst model assumes that all messages are sent in discreet communication phases where the application attempts to send as fast as the communication layer will allow. Under this model, the added gap, $\Delta g$, is incurred for each communication event. This second model again assumes that the applications runs at the speed of the processor sending $m$ messages, the maximum number of messages per processor from Figure 3.2, and would predict runtime, $r^{(b)}_{pred}$, as:

$$r^{(b)}_{pred} = r_{base} + m\Delta g$$

Application communication patterns determine which of the two models more closely predicts actual application runtime. The uniform model predicts that applications ignore increased gap until reaching a threshold equaling the application's average message interval. At this threshold, the applications should slowdown linearly with increasing gap. The burst model predicts a linear slowdown independent of average message interval.

### 3.2.3 Latency

Our model for latency is very simple. We simply model each synchronous Split-C read as the cost of an round trip, i.e. $4o + 2L$. Such a model, however, fails to account for any higher-level dependencies, such as synchronization via split-phased operations. Many of the programs have carefully orchestrated communication phases and these use a variety of synchronization mechanisms, including split phased reads, writes and barriers. The only application which performs many blocking reads is EM3D(read). Due to the low barrier frequency of most programs, we ignore the $L$ cost of barrier synchronization in our model.

We thus cannot expect a simple linear model for latency to provide much accuracy given the structure of most of the programs. The LogGP model certainly allows for more accurate models [39], however.

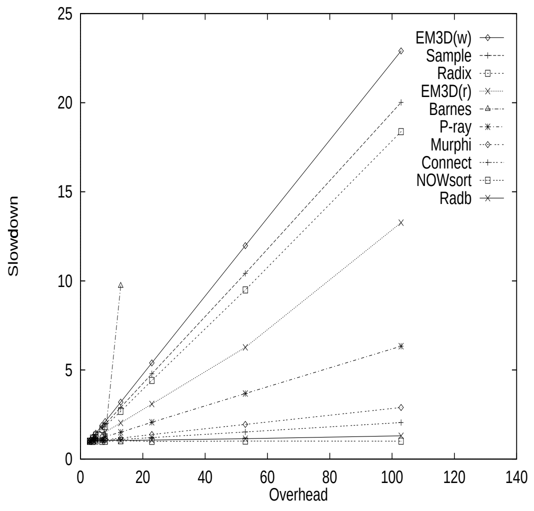
### 3.2.4 Gap

We model bulk transfers at costing a penalty of $o$ per 4 KB fragment plus a cost of $G$ per byte. However, few of the benchmarks use long messages. Thus, for this set of benchmarks, our predictions about the effects of $G$ may not be widely applicable. The NAS Parallel Benchmarks, described in the next chapter, use long messages exclusively and so it is there where we will explore sensitivity to $G$.

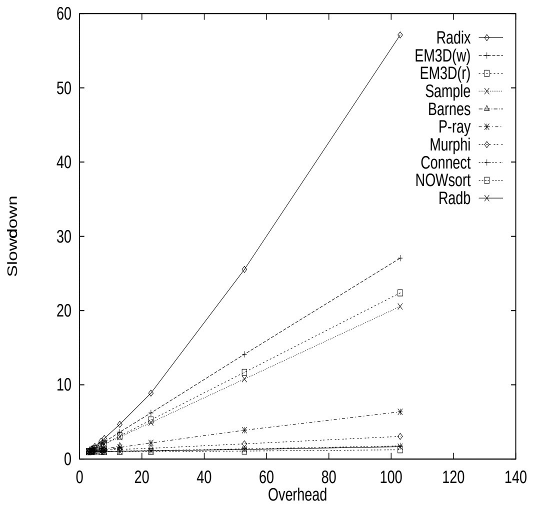
## 3.3 Sensitivity Results

Given our methodology and application characterization, we now quantify the effect of varying LogGP parameters on our application suite. To this end, we independently vary each of the parameters in turn to observe application slowdown. For each parameter, we attempt to explain any observed slowdowns based on application characteristics described in the last section. Using this intuition, we develop models to predict application slowdown.

Recall that any observed run time is a random variable, and thus any two run times, even very controlled conditions, will yield different results. In order to better account for these discrepancies, we take the minimum value of three sample run times. We take the minimum because it is most representative of the program execution behavior without outside interference. It was found that the majority of the discrepancy in run-time was due to effects of the GLUnix global control layer used to start and stop jobs [45]. These effects are not of interest to the results of this thesis. The data in this section shows that once the variance do to GLUnix is removed (by taking the minimum run-time), the Split-C/AM programs are very well behaved; programs do not yield widely varying results from run to run.



(a)



(b)

Figure 3.2: **Sensitivity to Overhead for 16 and 32 Nodes**
*This figure plots application slowdown as a function of overhead in microseconds. Slowdown is relative to application performance with our system's baseline LogGP parameters. Overhead is scaled by a factor of 30, from high-performance SAN protocols to high-overhead TCP/IP stacks. Measurements for the graph on the left were taken on 16 nodes, while measurements for the graph on the right were taken on 32 nodes with a fixed input size.*

### 3.3.1 Overhead

Figure 3.2(b) plots application slowdown as a function of added overhead measured in microseconds for our applications run on 32 nodes. The extreme left portion of the x-axis represents runs on our cluster. As overhead is increased, the system becomes similar to a switched LAN implementation. Currently, 100 $\mu$s of overhead with latency and gap values similar to our network is approximately characteristic of TCP/IP protocol stacks [60, 61, 103]. At this extreme, applications slow down from 2x to over 50x. Clearly, efforts to reduce cluster communication overhead have been successful. Further, all but one of our applications demonstrate a linear dependence to overhead, suggesting that further reduction in overhead will continue to yield improved performance. Qualitatively, the four applications with the highest communication frequency, Radix, Sample, and both EM3D read and write, display the highest sensitivity to overhead. Barnes is the only application which demonstrates a non-linear dependence to overhead. Instrumentation of Barnes on 16 nodes revealed that as overhead is increased, lock contention causes the program to go into livelock. With zero added overhead, the average number of failed lock attempts per processor is 2000 per timestep.

| **o $\mu$s** | **Radix** | | **EM3D(write)** | | **EM3D(read)** | | **Sample** | | **Barnes** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 2.9 | 7.8 | 7.8 | 38 | 38 | 114 | 114 | 13.2 | 13.2 | 43.2 | 43.2 |
| 3.9 | 10.5 | 10.3 | 48.1 | 47.5 | 138.7 | 130.7 | 16.1 | 15.8 | 50.1 | 44.9 |
| 4.9 | 13.2 | 12.9 | 58.1 | 57.0 | 161.6 | 147.3 | 18.7 | 18.4 | 56.3 | 51.8 |
| 6.9 | 18.7 | 18.0 | 77.4 | 76.1 | 208.8 | 180.5 | 23.8 | 23.6 | 76.1 | 60.3 |
| 7.9 | 21.5 | 20.5 | 87.4 | 85.6 | 232.9 | 197.2 | 26.5 | 26.2 | N/A | N/A |
| 13 | 36.3 | 33.3 | 138.5 | 133.3 | 354.4 | 280.3 | 39.3 | 39.1 | N/A | N/A |
| 23 | 68.9 | 58.9 | 236.2 | 228.6 | 600.1 | 446.7 | 65.2 | 65.0 | N/A | N/A |
| 53 | 198.2 | 135.7 | 535.9 | 514.5 | 1332.5 | 945.6 | 142.7 | 142.7 | N/A | N/A |
| 103 | 443.2 | 263.6 | 1027.8 | 991.0 | 2551.7 | 1777.2 | 272.1 | 272.2 | N/A | N/A |

| **o $\mu$s** | **P-Ray** | | **Mur$\varphi$** | | **Connect** | | **NOW-sort** | | **Radb** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 2.9 | 17.9 | 17.9 | 35.3 | 35.3 | 1.17 | 1.17 | 56.9 | 56.9 | 3.73 | 3.73 |
| 3.9 | 19.0 | 18.5 | 37.1 | 35.7 | 1.19 | 1.18 | 56.7 | 57.0 | 3.77 | 3.74 |
| 4.9 | 19.6 | 19.0 | 37.7 | 36.0 | 1.20 | 1.19 | 61.2 | 57.1 | 3.77 | 3.75 |
| 6.9 | 22.0 | 20.1 | 41.8 | 36.7 | 1.23 | 1.20 | 57.9 | 57.4 | 3.82 | 3.77 |
| 7.9 | 20.8 | 20.7 | 41.9 | 37.0 | 1.24 | 1.21 | 58.3 | 57.6 | 3.83 | 3.78 |
| 13 | 28.2 | 23.5 | 46.2 | 38.7 | 1.31 | 1.25 | 58.1 | 58.3 | 3.93 | 3.83 |
| 23 | 39.0 | 29.1 | 51.2 | 42.1 | 1.44 | 1.34 | 58.3 | 59.7 | 4.10 | 3.93 |
| 53 | 69.7 | 45.8 | 72.6 | 52.2 | 1.85 | 1.61 | 61.7 | 63.9 | 4.81 | 4.23 |
| 103 | 114.0 | 73.6 | 107.8 | 69.1 | 2.52 | 2.08 | 71.1 | 70.8 | 6.19 | 4.73 |

Table 3.3: **Predicted vs. Measured Run Times Varying Overhead**
*This table demonstrates how well our model for sensitivity to overhead predicts observed slowdown for the 32 node runs. For each application, the column labeled $measure$ is the measured runtime, while the column labeled $predict$ is the runtime predicted by our model. For frequently communicating applications such as Radix, EM3D(write), and Sample, the model accurately predicts measured runtimes.*

At 13 $\mu$s of overhead, the number of failed lock attempts per processor per timestep skyrockets to over 1 million. This implementation of Barnes does not complete for overhead values greater than 13 $\mu$s on 16 nodes and 7 $\mu$s on 32 nodes.

To determine the effect of scaling the number of processors on sensitivity to overhead, we executed our applications on 16 nodes with fixed inputs. Figure 3.2(a) plots the resulting slowdown as a function of overhead for runs on 16 nodes. With the exception of Radix, the applications demonstrate almost identical sensitivity to overhead on 16 processors as they did on 32 processors, slowing down by between a factor of between 2 and 25. Recall that Radix contains a phase to construct a global histogram. The number of messages used to construct the histogram is a function of the radix and the number of processors, not the number of keys. For a constant number of keys, the relative number of messages per processor increases as processors are added. Radix thus becomes more sensitive to overhead as the number of processors is increased for a fixed input size. In addition, the difference in sensitivities between 16 and 32 nodes is exacerbated by a serial phase in program, which is described below.

Table 3.3 describes how well our simple overhead model predicts application performance when compared to measured runtimes. For two applications which communicate frequently, Sample, and EM3D(write), our model accurately predicts actual application slowdown . For a number of other applications, most notably Radix, P-Ray and Mur$\varphi$, the sensitivity to overhead was actually stronger than our model's prediction; the model consistently under-predicts the run time, which is consistent with the assumptions made by the model.

Another way to describe the "serialization effect" is that our model implicitly assumes all work in the program is perfectly parallelizable; i.e., there are no serial phases in the program. This assumption leads to under-predicted run times. If a processor, $P_n$, serializes the program in a phase $n$ messages long, when we increase $o$ by $\Delta o$, then the serial phase will add to the overall run time by $n\Delta o$. However, the simple model does not capture the serialization effect when $P_m \neq P_n$.

A more important result of the of the serialization effect is that it reduces speedup as a function of overhead, i.e. speedup gets worse the greater the overhead. Thus, parallel efficiency will decrease as overhead increases for any applications which have a serial portion. Notice how for Radix, parallel decreases as a function of overhead when scaled from 16 to 32 nodes.

Radix sort demonstrates a dramatic example of the serialization effect. The sensitivity to overhead for Radix on 32 processors is over double that of 16 processors. When overhead rises to 100$\mu$s, the slowdown differential between 16 and 32 processors is a factor of three. The global histogram phase contains a serialization proportional to the radix and number of processors [39]. In the unmodified case, the phase accounts for 20% of the overall execution time on 32 processors. When the overhead is set to 100$\mu$s, this phase accounts for 60% of the overall execution time. However, on 16 processors with 100$\mu$s of overhead, the histogram phase takes only 16% of the total time.

In the case of P-ray, recall from Figure 3.1 that the application shows a strong communication imbalance. Three processors are "hot spots", likely containing popular objects. We conjecture that the nodes doing the most work are not the ones sending the most messages. By increasing the overhead, all other processors which attempt access to hotspots in the octree are likely slowed down by a linear factor not predicted by our model

The other five applications, however, actually demonstrate a stronger sensitivity to overhead than predicted by the model. The only application which demonstrates a non-linear dependence to overhead was Barnes. Further experimentation with the program revealed that as overhead was increased, lock contention caused the program to go into livelock. In fact, we were unable to add more than 10 $\mu$s of overhead and still have the application complete in a reasonable amount of time.

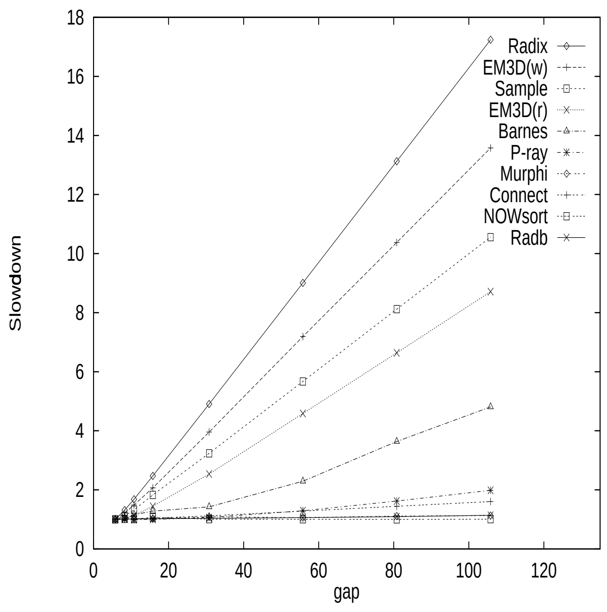


Figure 3.3: **Sensitivity to gap**
*This figure plots slowdown as a function of gap in microseconds. The gap is scaled by a factor of 20. While no hardware has a gap of over 20 $\mu$s, we can observe from the figure that many applications have linear responses to gap for then entire observed range.*

### 3.3.2 gap

We next measure application sensitivity to gap. Figure 3.3 plots application slowdown as a function of added gap in microseconds. The programs demonstrate widely varying reactions to gap, ranging from being unaffected by 100 $\mu$s of gap to slowing down by a factor of 16. The qualitative difference between application sensitivity to gap and sensitivity to overhead can be explained by the fact that sensitivity to gap is incurred by the program only on the portion of the messages where the application attempts to send at a rate greater than the gap. The rest of the messages are not sent quickly enough to be affected by the gap. Thus, infrequently communicating applications can potentially ignore gap entirely, while overhead is incurred independent of message frequency. The four applications with the highest communication frequency, Radix, EM3D(write) and read, and Sample, suffer the largest slowdowns from added gap. The other applications are much more tolerant to gap, slowing down by no more than a factor of 4 even in the presence of 100 $\mu$s of added gap.

Given the linear dependence to gap demonstrated by the applications in Figure 3.3, we believe that for our applications, the burst model more accurately predicts application behavior. Table 3.4 depicts how well the burst model predicts actual program slowdown. As anticipated, the

| **g $\mu$s** | **Radix** | | **EM3D(write)** | | **EM3D(read)** | | **Sample** | | **Barnes** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 5.8 | 7.8 | 7.8 | 38 | 38 | 114 | 114 | 13.2 | 13.2 | 43.2 | 43.2 |
| 8 | 10.2 | 11 | 46.1 | 49.9 | 119 | 134.8 | 14.8 | 16.5 | 44.1 | 45.4 |
| 10 | 13 | 14.2 | 56.5 | 61.8 | 129.7 | 155.6 | 17.5 | 19.7 | 50.2 | 47.5 |
| 15 | 19.2 | 20.5 | 78.5 | 85.6 | 164.7 | 197.2 | 24.2 | 26.2 | 55.3 | 51.8 |
| 30 | 38.1 | 39.7 | 150.3 | 157.1 | 289.3 | 321.9 | 42.9 | 45.6 | 61.6 | 64.6 |
| 55 | 69.9 | 71.7 | 273.1 | 276.2 | 523 | 529.8 | 75.1 | 78 | 99.1 | 85.9 |
| 80 | 101.9 | 103.7 | 394 | 395.4 | 756.9 | 737.7 | 107.5 | 110.4 | 157.3 | 107.2 |
| 105 | 133.8 | 135.7 | 515.6 | 514.5 | 993.1 | 945.6 | 139.7 | 142.7 | 207.9 | 128.5 |

| **g $\mu$s** | **P-Ray** | | **Mur$\varphi$** | | **Connect** | | **NOW-sort** | | **Radb** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 5.8 | 17.9 | 17.9 | 35.3 | 35.3 | 1.17 | 1.17 | 56.9 | 56.9 | 3.73 | 3.73 |
| 8 | 18.1 | 18.6 | 37.4 | 35.8 | 1.19 | 1.18 | 57.9 | 57.0 | 3.77 | 3.74 |
| 10 | 17.8 | 19.3 | 36.1 | 36.2 | 1.21 | 1.19 | 57.6 | 57.2 | 3.78 | 3.75 |
| 15 | 17.9 | 20.7 | 36.2 | 37.0 | 1.24 | 1.23 | 60.9 | 57.6 | 3.80 | 3.78 |
| 30 | 19.1 | 24.9 | 38.4 | 39.5 | 1.34 | 1.32 | 57.3 | 58.6 | 3.86 | 3.85 |
| 55 | 23.2 | 31.8 | 37.5 | 43.8 | 1.51 | 1.50 | 57.2 | 60.4 | 3.96 | 3.98 |
| 80 | 29.0 | 38.8 | 39.3 | 48.0 | 1.68 | 1.69 | 56.9 | 62.1 | 4.08 | 4.10 |
| 105 | 35.5 | 45.8 | 39.9 | 52.2 | 1.85 | 1.88 | 57.4 | 63.9 | 4.25 | 4.23 |

Table 3.4: **Predicted vs. measured run times varying gap**
*This table demonstrates how well the burst model for sensitivity to gap predicts observed slowdown. For each application, the column labeled $measure$ is the measured runtime, while the column labeled $predict$ is the runtime predicted by our model.*

model over predicts sensitivity to gap since not all messages are sent in bursts. The model works best for heavily communicating applications, as a larger percentage of their messages are slowed by gap.

The two models considered demonstrate a range of possible application behavior. Applications communicating at very regular intervals would follow the uniform model, while applications communicating in discreet phases would track the burst model.

### 3.3.3 Latency

Traditionally, most attempts at improving network performance have focused on improving the network latency. Further, perceived dependencies on network latencies have led programmers to design their applications to hide network latency. Figure 3.4 plots application slowdown as a function of latency added to each message. Perhaps surprisingly, most of applications are fairly insensitive to added latency. The applications demonstrate a qualitatively different ordering of sensitivity to latency than to overhead and gap. Further, for all but one of the applications the sensitivity does not appear to be strongly correlated with the read frequency or barrier interval, the operations most likely to demonstrate the strongest sensitivity to latency.

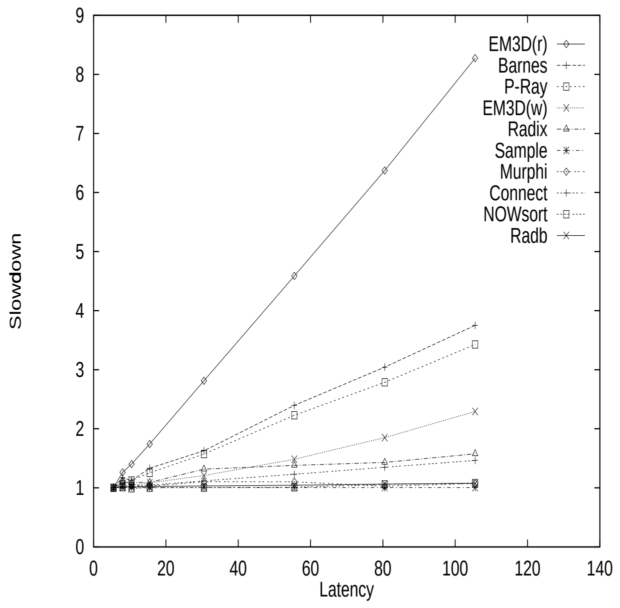
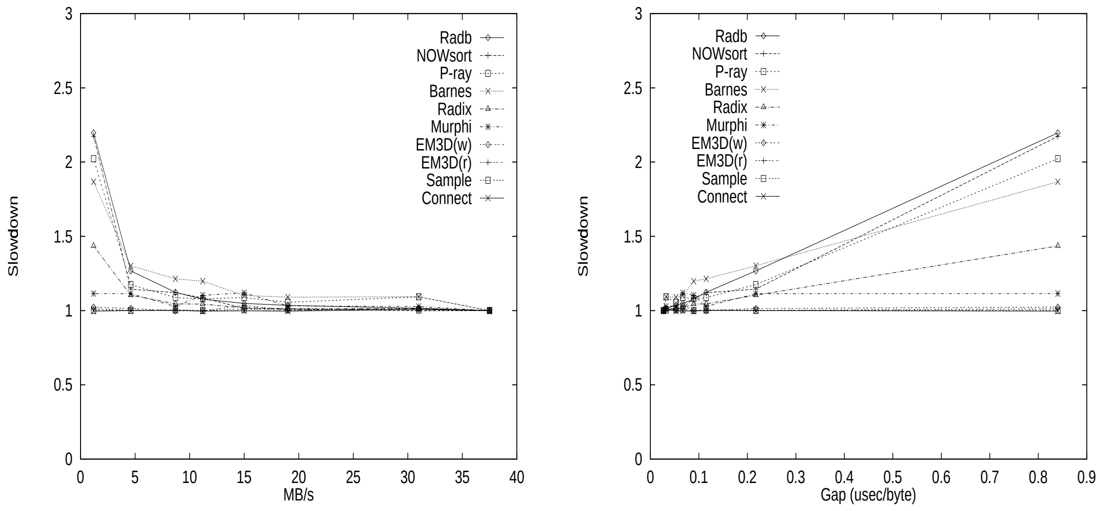


Figure 3.4: **Sensitivity to Latency**
*This figure plots slowdown as a function of latency in microseconds. Latency is scaled by a factor of 20, ranging from SAN class latencies to a network built out of several long latency ATM switches. The figure shows the effectiveness of a wide range of practical latency tolerating techniques.*

The sensitivity of EM3D(read), Barnes, P-Ray, and Connect to latency results from these applications' high frequency of read operations (see Figure 3.2). Read operations require network round-trips, making them the most sensitive to added latency. However, for all but one of these applications, the observed slowdowns are modest (at most a factor of four in the worst-case) even at the latencies of store-and-forward networks (100 $\mu$s).

EM3D(read) performs a large number of blocking reads; it represents a "worst-case" application from a latency perspective because it does nothing to tolerate latency. It is also the only application for which a simple model of latency is accurate. Interestingly, for equal amounts of added "work" per message (100 $\mu$s of latency and 50 $\mu$s of overhead), the simple latency model for EM3D(read) is quite accurate yet the simple overhead model under predicts the run time.

The applications which do not employ read operations largely ignore added latency. The small decrease in performance at the tail of the slowdown curves is caused by the increase in gap associated with large latencies as the Active Message flow control mechanism limits the network capacity (see Table 2.2).



(a) (b)

Figure 3.5: **Sensitivity to Bulk Gap**
*This figure plots slowdown as a function of maximum available network bandwidth (a) as well as bulk Gap (b). Bandwidth is scaled from 10 Mb Ethernet speeds to SAN speeds of near 30 MB/s. Although bandwidth is a more intuitive measure, it is difficult to visualize the sensitivity to bandwidth from figure (a) because as we scale $G$ in a linear manner we are plotting a $\frac{1}{x}$ bandwidth curve. Figure (b) shows some applications slow down in a linear fashion as we scale $G$ linearly.*

### 3.3.4 Bulk Gap

Only applications attempting to send large amounts of data in bursts should be affected by reductions in bulk transfer bandwidth. Note that we do not slow down transmission of small messages, but rather add a delay corresponding to the size of the message for each bulk message. Further, applications should tolerate decreases in available bulk bandwidth until the bandwidth dips below the application's requirements at any point during its execution.

Figure 3.5(a) plots application slowdown as a function of the maximum available bulk transfer bandwidth, a more intuitive measure than Gap. We also plot the same sensitivity to Gap in Figure 3.5(b) in order to show the linear relationship between slowdown and $G$. Overall, the applications in our suite do not display strong sensitivity to bandwidth. No application slows by more than a factor of three even when bulk bandwidth is reduced to 1 MB/s. Further, all of the applications, including Radb, which moves all of its data in a single burst using bulk messages, do not display sensitivity until bulk bandwidth is reduced to 15 MB/s. Surprisingly, the NOW-sort is also insensitive to reduced bandwidth. This version of the NOW-sort uses two disks per node. Each disk can deliver 5.5 MB/s of bandwidth [8], and during the communication phase a single disk is used for reading and the other for writing. As Figure 3.5 shows, NOW-sort is disk limited. Until the network bandwidth drops below that of a single disk, NOW-sort is unaffected by decreased bandwidth.

## 3.4 Summary

Varying the LogGP parameters for our network of workstations and benchmark suite led to a number of interesting results. We organize these around our four areas of contributions.

### 3.4.1 Performance Analysis

In the performance analysis area, we find that our apparatus was quite effective in measuring the sensitivities of the Split-C/AM suite. Running real programs on the apparatus was not a problem. All the programs were well behaved, giving consistent results from run to run. Even Barnes, which would go into live-lock as we scaled overhead, exhibited this behavior in a repeatable way.

We were able to observe a number of effects that would be difficult to observe using other techniques. These include a hyper-sensitivity to overhead from a number of applications. In addition, we were able to observe live-lock effects for Barnes which would require fairly sophisticated models or simulators to observe. Our apparatus also allows us to quantify the point in the overhead space where this effect occurs.

### 3.4.2 Application Behavior

In the behavior area, we find that applications displayed the strongest sensitivity to network overhead, slowing down by as much as a factor of 50 when overhead is increased to roughly 100 $\mu$s. Even lightly communicating processes suffer a factor of 3-5 slowdown when the overhead is increased to values comparable to many existing LAN communication stacks. Frequently communicating applications also display strong sensitivity to gap suggesting that the communication phases are bursty and limited by the rate at which messages can be injected into the network.

The effect of added latency and bulk gap is qualitatively different from the effect of added overhead and gap. For example, applications which do not perform synchronization or read operations (both of which require round trip network messages) can largely ignore added latency. For our measured applications, the sensitivity to overhead and gap is much stronger than sensitivity to latency and per-byte bandwidth.

### 3.4.3 Network Architecture

The most interesting result, which relates to the architecture area, is the fact that all the applications display a linear dependence to both overhead and gap. This relationship suggest that continued architectural improvements in these areas should result in a corresponding improvement in application performance (limited by Amdahl's Law). In contrast, if the network performance were "good enough" for the applications, (i.e., some other part of the system was the bottleneck), then we should observe a region were the applications did not slow down as network performance decreased. In contrast, efforts in improving network latency will not yield as much performance improvements across as wide a class of applications.

A second architectural result is that there is an interesting tradeoff between processor performance and communication performance. For many parallel applications, relatively small improvements in network overhead and gap can result in a factor of two performance improvement. This result suggests that in some cases, rather than making a significant investment to double a machine's processing capacity, the investment may be better directed toward improving the performance of the communication system.

### 3.4.4 Modeling

In the modeling area, we found that for both overhead and gap, simple models are able to predict sensitivity to these parameters for most of our applications. The effects of latency, on the other hand, are harder to predict because they are more dependent on application structure. The applications used a wide variety of latency tolerating techniques, including pipelining (radix, sample), batching (EM3D, Radb), caching (Barnes, P-ray) and overlapping (Mur$\varphi$ and NowSort). Each of these techniques requires more sophisticated models to capture the effect of added latency than our frequency-cost model allows.

# Chapter 4

# NAS Parallel Benchmark Sensitivity

*... it is a consistent theme that each generation of computers obsoletes the performance evaluation techniques of the prior generation*. — Hennessey & Patterson, Computer Architecture: A Quantitative Approach

The NAS Parallel Benchmarks (NPB) are widely used to evaluate parallel machines. To date, every vendor of large parallel machines has presented NPB version 1.0 results [10]. The recent convergence of parallel machines and the introduction of a standard programming model (MPI), the NAS group created version 2.2 of the benchmark suite [11]. In contrast to the vendor-specific implementations of version 1.0, NPB 2.2 presents a consistent, portable, and readily available workload to parallel machine designers, analogous to the SPECcpu benchmarks for single-processor machines. Much has been written about the theoretical techniques in these codes [13, 111], but an understanding of their practical communication behavior is at best incomplete.

In this chapter, we examine the sensitivities of three of the six NAS parallel benchmarks: **FT**, **IS** and **MG**. The three are computational kernels from numerical aerodynamic simulation codes. The other three benchmarks, **SP**, **BT** and **LU**, are longer codes that are considered pseudo-applications. Unfortunately, apparatus limitations did not allow us to run the much larger pseudo-applications for this study. The input set comes in 3 sizes: class A, B, and C. All our experiments are run on the class B size, which is appropriate to run on 32 nodes but does not scale down to single node sizes. However, because we are not measuring the scalability of the codes class B is reasonable input set size to run on our apparatus.

| Program | Run Time (sec) | Collective All-to-All(v) | | MPI Device Level | | Active Message Level | | Max-Min Ratio |
|---|---|---|---|---|---|---|---|---|
| | | Msgs. | Bytes | Msgs. | Bytes | Small | 4K Frag | |
| FT | 173.2 | 20 | 325058560 | 660 | 325058560 | 1980 | 79360 | 0.0% |
| IS | 18.7 | 20 | 40833600 | 670 | 40833600 | 2010 | 9969 | 17.0% |
| MG | 17.8 | 2854 | 27570880 | 2854 | 27570880 | 8562 | 6731 | 0.1% |

Table 4.1: **NPB Communication Summary**
*For a 32 processor configuration, the table shows run times, the number and size of collective operations at the MPI level, the maximum number and size of operations at the MPI Device level (per processor), the resulting number of messages at the Active Message level (per processor), and the percentage skew, measured in bytes, between the processors that sent the maximum and minimum number of bytes.*

## 4.1 Characterization

In this section we characterize the NPB much in the same way as we did the Split-C/AM benchmarks in the previous chapter. We begin with a brief description of each benchmark, followed by a balance graph and message count analysis. We compare the communication characteristics of the benchmarks to the Split-C/AM programs. The benchmarks are:

- **FT**: This kernel performs a 3-D Fast Fourier Transform (FFT) on a $256^2 \times 512$ grid. The program implements the FFT as a series of 1-D FFTs. Each iteration, the program does a global transpose of all the data, thus performing a perfectly balanced all-to-all communication pattern. The FFT requires a large amount of computation in addition to a large volume of communication.

- **IS**: The Integer Sort (IS) benchmark performs a bucket sort on 1 million 32 bit keys per processor. Notice that the run time for **IS** is much higher than for the equivalent Split-C sorts [39]. Like sample sort, this sort performs unbalanced all-to-all communication, relying on a random key distribution for load balancing. Thus, the communication pattern is dependent on the data-set.

- **MG**: This program solves a poisson equation on a $256^3$ grid using a multigrid "W" algorithm. Unlike FT and IS, communication is quite localized, occurring between neighboring processors on the different grid levels. The communication pattern does not depend on the data.

Figure 4.1 shows that the three NPB codes are well structured. **FT** and **MG** are perfectly balanced; each processor sends and receives the same amount of data. **IS** is slightly unbalanced,

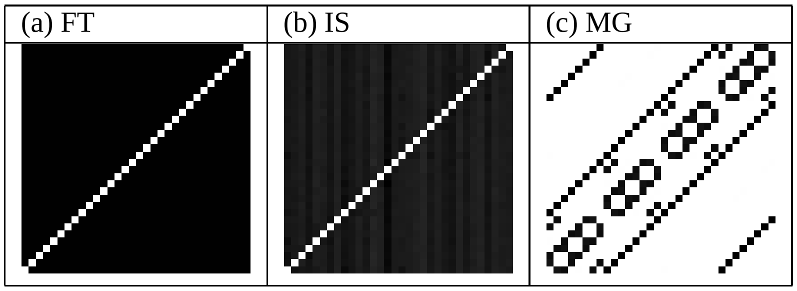


Figure 4.1: **NAS Parallel Benchmarks Communication Balance**
*This figure demonstrates the communication balance between each of the 32 processors for 3 of the NAS Parallel Benchmarks. The greyscale for each pixel represents a byte count, as opposed to the message counts in Figure 3.1. Each application is individually scaled from white, representing zero bytes, to black, representing the maximum byte count per processor as shown in Table 4.1. The $y$-coordinate tracks the message sender and the $x$-coordinate tracks the receiver.*

but on the whole is not overly imbalanced. For **FT** and **IS**, Figure 4.1 shows that communication is global, which each processor sending much data to all other processors. Such patterns will stress the bisection bandwidth of the network. We can see that the hierarchical grid pattern in **MG** results in a much more localized communication pattern.

Table 4.1 shows just how different the NPB are from the Split-C/AM applications in Table 3.2. First, for **FT** and **IS**, we see that all communication is performed at the MPI level in collective operations. Using an MPI collective operations allows the communication layer to perform a number of optimizations, for example inserting global barriers [19] or using pairwise exchanges [39] that can greatly improve the performance of such operations. However, the MPICH implementation reduces these operations to a series of simple point-to-point sends at the MPID level. Table 4.1 shows that even after this reduction, the number of messages is still very small. For example, the **IS** sort sends only 670 messages per processor, compared with over 1 million for the Split-C sample sort.

Clearly, the authors of the NPB have taken pains to minimize message cost by aggregating messages at the application level. Table 4.1 shows that as a result, the very few messages sent are very large. For **FT**, the result is that on a 32 processor system the messages are half a megabyte each. For **IS**, the size is still quite large, averaging about 60 KB. In both these applications, the median message size is close to the average.

**MG**, however, is different in that most messages are small; the median message size occurs at 32 KB, but the average message size is 10 KB. Figure 4.2 plots a histogram of message sizes. It shows that the vast majority of the data is sent in large messages; 97% of the data sent in messages

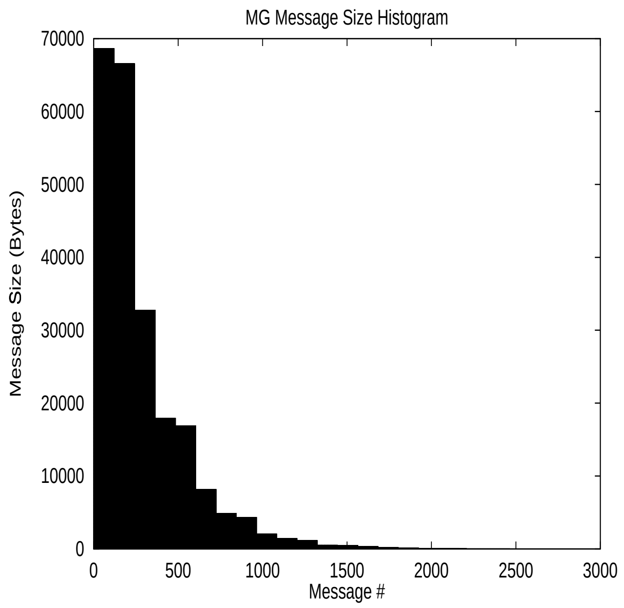


Figure 4.2: **MG Message Size Histogram**
*This figure shows the message size histogram for all 2615 messages sent by a single processor while running the MG benchmark for class B input on 32 nodes. Message sizes are sorted in decreasing order. Notice that message sizes follow an exponential distribution. Although most messages are small, nearly all the data is sent in very large messages.*

over 4 KB and 99% sent in messages over 1 KB. Although it has a larger number of small messages, the total message cost is dominated by the large messages. The small message count is so low (around 1000) that these fail to have much impact on the total communication time.

## 4.2 Sensitivity Results

In this section, we examine the sensitivity of the NPB to the LogGP parameters. In particular, we concentrate the discussion on $o$ and $G$. Given our MPI-GAM apparatus, and the nature of the NPB sending a few long messages, these are likely to be the most important parameters. Recall that for long messages, our apparatus inflates $o$ for every 4KB fragment. Long messages will be affected by $o$, although not nearly to the same extent as small messages. We will examine the validity of this approach in the discussion section.

We attempt to explain any slowdown using the our knowledge of the MPI layer and the simple piecewise-linear model of message passing in Section 2.3.2. Where we can not explain the discrepancy, we hypothesize on reasons why a noticeable discrepancy exits between the measured

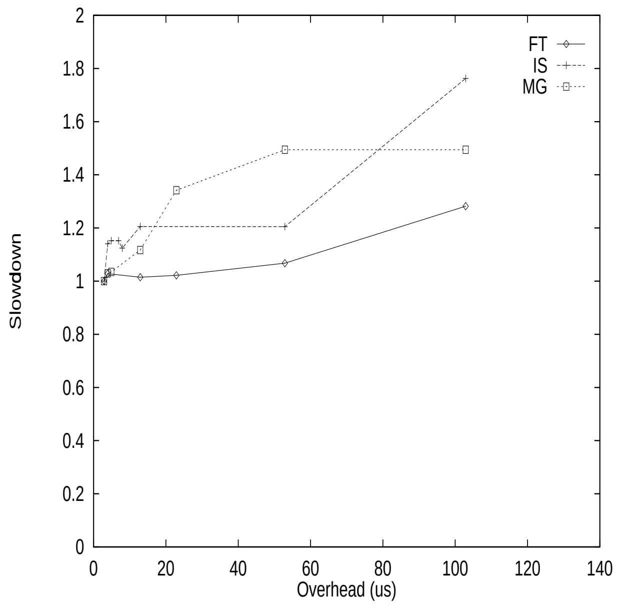


Figure 4.3: **NPB Sensitivity to Overhead**
*This figure plots measured slowdown as a function of overhead in microseconds.*

and predicted performance.

As with the Split-C section, each plotted data point is the minimum of three runs of the program. However, we shall see from the data, the NPB are not as well behaved as the Split-C/AM programs. In addition, there were bugs in the network switches that made data collection impossible for some of the benchmarks. [1]

## 4.2.1 Overhead

Given the MPICH design and our empirical apparatus, we would expect modest sensitivities to overhead. Figure 4.3 shows the slowdown of the three benchmarks as we scale the AM layer overhead from 5 to 100 $\mu$s. The first noticeable result is that the NPB are much less sensitive to overhead than the Split-C/AM programs. Instead of slowing down by *factors* of 20 or 30, as most of the Split-C/AM programs are, the NPB only slow down by factors of less than 2. Indeed, the program with the heaviest volume of communication, **FT**, is only slowed down a modest 20% at 100 $\mu$s of overhead. Also, unlike many Split-C/AM benchmarks, there are noticeable flat regions; **IS** and **FT** have noticeable flat regions out to 60 $\mu$s while **MG** has a flat region past 60 $\mu$s.

[1] The workaround from the vendor was to increase all messages sizes to $>1$ KB. While allowing the NPB to run without crashing, this would artificially inflate the gap, Gap and latency parameters of the study to unacceptable levels. Corrected switches were not available in time for this thesis.

| **o** $\mu$**s** | **FT** | | **IS** | | **MG** | |
|---|---|---|---|---|---|---|
| | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 10 | 173.3 | 173.3 | 18.7 | 18.7 | 17.8 | 17.8 |
| 11 | 178.0 | 173.3 | 21.3 | 18.7 | 18.3 | 17.8 |
| 20 | 175.7 | 174.8 | 22.5 | 18.9 | 19.8 | 18.0 |
| 30 | 177 | 176.4 | - | - | 23.8 | 18.3 |
| 60 | 184.9 | 181.2 | 22.6 | 19.8 | 26.5 | 19.0 |
| 110 | 222.0 | 189.3 | 32.9 | 20.9 | 26.6 | 20.2 |

Table 4.2: **NPB Predicted vs. Measured Run Times Varying Overhead**
*This table demonstrates how well our model for sensitivity to overhead predicts observed slowdown for the 32 node runs. For each application, the column labeled $measure$ is the measured runtime, while the column labeled $predict$ is the runtime predicted by our model. The model accurately predicts measured run times for smaller overheads.*

Table 4.2 shows the measured vs. predicted performance for the NPB. The models are reasonably accurate for $o$ values under 50 $\mu$s. But for high overheads, the applications are more sensitive than the models predict. The extra sensitivity might be because a round-trip is required for most messages. At very high overheads (e.g. 100 $\mu$s ) contention effects may dominate, magnifying the cost of the round-trip set-up.

### 4.2.2 gap

Figure 4.4 shows the NPB are much less sensitive to gap than to overhead. One benchmark, **FT**, can ignore a $g$ entirely! Although messages are sent in bursts, the very large size of most messages means that even a high $g$ can be amortized over a low communication frequency. A safe conclusion that can be drawn is that these benchmarks are very insensitive to $g$. This is in contrast to many of the Split-C/AM benchmarks, which exhibited a strong sensitivity to $g$.

### 4.2.3 Latency

As we would expect, the NPB are quite insensitive to increased $L$. Figure 4.5 plots the latency figures for two of the benchmarks. Unfortunately, a bug in the switches prevented collecting data for the **FT** benchmark. It is a relatively safe assumption, given our characterization, that **FT** would have a very low sensitivity to $L$. Recall that a round trip is only required once per message at the MPID level, so the number of round trips numbers in the 100's per processor.

Even with the MPICH-GAM round-trip set-up cost, the applications' infrequent use of messages and the very large size of most messages allows them to amortize the cost of a long $L$.

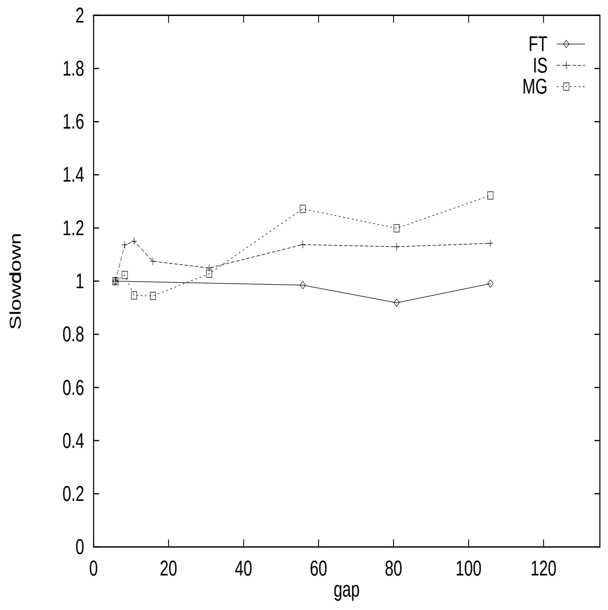


Figure 4.4: **NPB Sensitivity to gap**
*This figure plots measured slowdown as a function of gap in microseconds.*

Given that GAM provides in-order delivery, it would not bee too difficult to remove this round-trip set-up. In that case we would expect sensitivity to $L$ to be even lower than the results presented here.

### 4.2.4 Bulk Gap

We now turn our attention to bulk Gap. Figure 4.6 plots sensitivity to bandwidth, or $\frac{1}{G}$. We see that there is a flat region to about 5 MB/s and then a sharp turn upward. The first result is that the NPB are quite insensitive to a bandwidth below 12.5 MB/s. This is an important result because 100 Mb Ethernet and 155 Mb ATM are at or well above this performance level. Thus, these technologies would be adequate for running the NPB on the class of processor used in this study.

The sharp inflection, however, demands more attention. An important question is if the infection is due to an abnormal rise in sensitivity. Recall we are plotting sensitivity to a non-linear change in the independent variable, i.e., sensitivity as a function of $\frac{1}{x}$. We are scaling $G$ in a linear fashion but plotting bandwidth instead, which is a non-linear change. It is thus difficult to ascertain via the naked eye if the inflection is due to a hyper-sensitivity or simply following a normal $\frac{1}{x}$ curve.

In order to ascertain the sensitivity, we turn to the linear model of MPI performance developed in Section 2.3.2. Recall that in that section, we described how MPI performance should be affected by inflation of the LogGP parameters. Table 4.3 shows the resulting measured vs. predicted

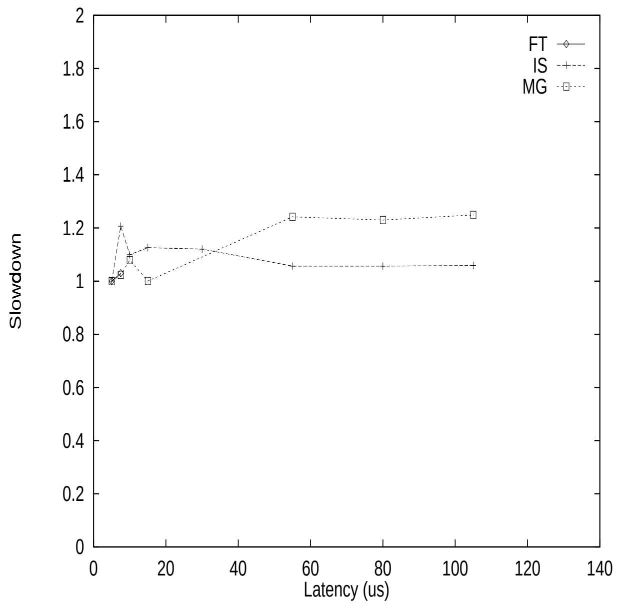


Figure 4.5: **NPB Sensitivity to Latency**
*This figure plots measured slowdown as a function of latency in microseconds.*

performance as we scale $G$. We see that indeed, at 1.2 MB/s the applications are hyper-sensitive to $G$, in that they take much longer to run than predicted by the model. The cause of the increased slowdown is unclear; the LANai can continue to receive messages so network congestion should not be a problem. Whatever the result of the hyper-sensitivity, the result shows a technology such as switched 10Mb Ethernet is probably not sufficient for these applications.

## 4.3 NPB Sensitivity Summary

As with the Split-C/AM chapter, we organize our conclusions on the NPB around the areas of performance analysis, application behavior, architecture and modeling.

### 4.3.1 Performance Analysis

In the performance analysis regime, we find that our apparatus is adequate for measuring sensitivity of the NPB, although the programs are somewhat noisier than the Split-C programs. The apparatus also showed that it can measure non-linear responses for a number of the NPB. The extra layer of communication protocol between our apparatus and the application (MPI), did cause us to expend extra effort in modeling that we did not do for the Split-C/AM programs. A method around

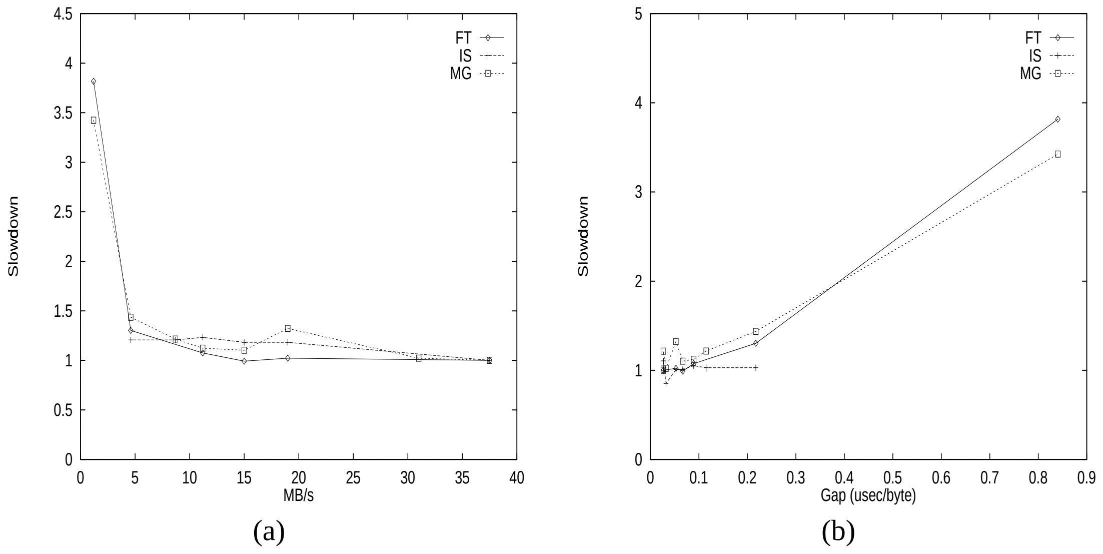


Figure 4.6: **NPB Sensitivity to Bandwidth and Gap**
*This figure plots slowdown as a function of maximum available network bandwidth (a) as well as Gap (b). Bandwidth is a more intuitive measure, but it is difficult to visualize the sensitivity to bandwidth from the Figure (a) because as we scale $G$ in a linear manner we are plotting a $\frac{1}{x}$ bandwidth curve. Figure (b) shows* **FT** *and* **MG** *slow down in a hyper-linear fashion as we scale $G$ linearly.*

this problem would be to add delays into the MPI layer directly.

As a side effect of slowing down GAM instead of MPI, the apparatus introduces a somewhat artificial sensitivity to $o$. For example, many machines, (e.g., Cray T3D, Intel Paragon, Meiko CS-2 [7, 31]), do not introduce extra overhead on a per-fragment basis. However, a number of TCP/IP stacks do exhibit per-fragment overheads [26, 60, 61]; such per-fragment overheads form a visible "sawtooth" line in per-byte costs. Our per-fragment overhead is thus a reasonable, if somewhat exaggerated, approximation of this class of software overhead.

### 4.3.2 Application Behavior

Turing our attention to the application behavior area, the NPB we examined in this chapter have a communication structure that is dominated by infrequent, large and bursty communication. The resulting sensitivity to $G$ is quite intuitive. They are also sensitive to $o$ somewhat, although our results are somewhat inflated due to the apparatus construction mentioned previously. Sensitivity to $g$ was almost very low, even non-existent for **FT**.

The structure of the NPB shows their origins quite clearly: the codes were developed on

| **BW(1/G)** | **FT** | | **IS** | | **MG** | |
|---:|---:|---:|---:|---:|---:|---:|
| (MB/s) | *measure* | *predict* | *measure* | *predict* | *measure* | *predict* |
| 37 | 173.3 | 173.3 | 18.7 | 18.7 | 17.8 | 17.8 |
| 19 | 177.0 | 181.6 | 22.1 | 19.7 | 23.5 | 18.5 |
| 15 | 171.8 | 186.2 | 19.6 | 18.8 | 19.5 | 18.9 |
| 11.2 | 186.2 | 193.5 | 23.0 | 21.2 | 19.9 | 19.5 |
| 4.6 | 225.6 | 235.2 | 22.6 | 26.4 | 25.5 | 23.0 |
| 1.2 | 660.9 | 437.6 | - | - | 60.8 | 40.2 |

Table 4.3: **NPB Predicted vs. Measured Run Times Varying Bulk Gap**
*This table demonstrates how well our model for sensitivity to Bulk Gap predicts observed slowdown for the 32 node runs. For each application, the column labeled* $measure$ *is the measured runtime, while the column labeled* $predict$ *is the runtime predicted by our model. The model accurately predicts measured run times for bandwidths greater than 10Mb/s.*

MPP machines as the iPSC and Delta which have very high message passing costs [32]. The high communication costs, yet ample bandwidth, on these machines lead to a design where communication is avoided as much as possible. When communication is necessary, it is packed into a few large messages. Given this history, the rather low sensitivities should not be too surprising.

It is instructive to compare the structure and resulting sensitivity of the NPB codes to the Split-C/AM codes. The Split-C/AM were developed in the context of very low-overhead machines, e.g., the CM-5, and Berkeley NOW. Thus, these programs assumed low-overhead and so show quite a high sensitivity to it. This raises the a classic engineering analysis question: are our results merely the result of a historic accident, or is there something more fundamental going on? We shall explore this question in greater detail in Chapter 7.

### 4.3.3 Network Architecture

Our architectural conclusions for the NPB are rather meager. Primarily, they are sensitive to per-byte network bandwidth. In particular, as we saw from the communication balance graphs, a machine's bisection bandwidth will be an issue for two of these benchmarks. However, unlike the Split-C/AM applications, communication is so infrequent on the machine sizes studied that for the NPB, improvements in per-node processor performance, as opposed to network performance, will yield the largest benefits.

### 4.3.4 Modeling

In the modeling area, we find that the 3 benchmarks exhibit stronger sensitivities to low-performance networks than simple models describe. However, this should not cause too much concern. The fact that the models fail to describe this class of low performance networks should not be surprising given that we are scaling the apparatus by an order of magnitude. The only conclusion one can draw is that in spite of their highly optimized communication, very cheap, low performance networks are not suitable for the NPB.

# Chapter 5

# NFS Sensitivity

*... but just running a lot of simulations and seeing what happens is a frustrating and finally unproductive exercise unless you can somehow create a "model of the model" that lets you understand what is going on.* — Paul Krugman, from a talk given to the European Association for Evolutionary Political Economy.

In this chapter, we examine the sensitivity of Sun's Network File System (NFS) to network performance. Our motivation is driven by the fact that previous work shows that 60%-70% of LAN traffic is filesystem related [48, 76]. We apply the same basic methodology used in the previous two chapters. The NFS application parameter space, however, is much larger than he Split-C/AM programs or the NPB. In the previous two chapters, run-time was the simple figure of merit. In the NFS case, there can be many different metrics, e.g. read bandwidth, write bandwidth, and response time.

Our method of fixing the class of inputs that are the traditional characteristics of NFS workloads, e.g. the mix of reads/writes/lookups, is to use the SPECsfs benchmark [94]. The SPECsfs benchmark is an industry-standard benchmark used to evaluate NFS servers. The networking parameters are the same LogGP parameters used throughout this thesis.

The output of the SPECsfs benchmark is a two-dimensional curve of response time vs. throughput, for a fixed mix of operations, as opposed to a point-metric such as run-time. Because of the two-dimensional nature the SFS curve, our results are presented differently than in previous chapters. Instead of a fixed slowdown line, the results are three dimensional: throughput vs. response time vs. change in network performance. While we could plot a single 3-D graph of these parameters, it is more informative to plot a series of 2-D graphs. The SFS curve has important structures that would be difficult to discern in a single 3-D graph. We detail the important parts of the

SFS curve in the next section.

In order to understand the SFS curve, a more complex model of the NFS system is necessary than the simple frequency-cost pairs used in the Split-C/AM and NPB models. We thus use simple queuing theoretic models to understand the relationship between the networking parameters and the SFS curve. As in previous chapters, our goal in this work is not to develop highly accurate models. Rather, the purpose of the queuing model is to gain insight as to how a system should behave as we change the networking parameters. The model's value lies in its ability to identify *why* the system responds as it does. The quote at the beginning of this chapter illustrates this same purpose of analytic models in the field of economics.

The combined use of a model and experimental data forms a synergy which is much more powerful than either alone. With only a model, we can predict the responses but the results are suspect. Measured data alone, while not suffering from lack of credibility, often lacks the simple conceptual interpretations that models provide. Using both a model and measurement we can explain the measured results in terms of the model. Points where the data deviates from model predictions expose weaknesses in our understanding of the system.

A side benefit of our choice of workload, SPECsfs, is that it allows us to compare our results to industry published data. More importantly, using the techniques in this work we can infer the structure of the NFS servers from the data published by SPEC.

The remainder of the chapter is organized as follows. We first describe the details of the apparatus relevant to NFS servers in Section 5.1. Section 5.2 provides background on SPECsfs, the workload for this chapter. Next, Section 5.3 introduces our simple queuing-theoretic model of an NFS system. Section 5.4 documents previous work on NFS. Next, Section 5.5 documents the measured sensitivities to network parameters on two live systems and describes the accuracy of the predictions made by the model. Section 5.6 summarizes some implications of our results, and analyzes industrial data in the context of our methods.

## 5.1 Experimental Setup

This section introduces the experimental set-up used in our NFS experiments. We focus on the disk sub-system because this is the component of interest unique to NFS performance. Recall that the networking apparatus used to control the LogGP parameters was described in detail Section 2.3.3. This section first describes the general characteristics of the clients, followed by a more detailed description of the two different servers.

As in the Split-C/AM and NPB experiments, all of the machines in our experiments consist of Sun Ultra-1 workstations. Attached to the S-Bus is an internal narrow SCSI bus and local disk that holds the operating system and swap space. All the clients have 128 MB of main memory. We use a total of 4 clients: 3 load-generators and 1 master control station. The control station and other machines are also connected via a switched 10Mb/s Ethernet. The Ethernet is used to start and stop the benchmark as well as monitor results.

The primary difference between our two servers is the disk sub-system. The "SCSI" system contains 128MB of main memory and 24 7200 RPM 9GB IBM drives. The drives are evenly divided between two SCSI buses. The S-bus interfaces used are the fast-wide Sun ''FAS" controller cards. In contrast, the "RAID" system contains 448 MB of main memory. The 28 7200 RPM 9GB Seagate disks are contained in a Sun "A3000" RAID. The disks are divided into 5 RAID level-0 (striping) groups; 4 groups have 6 disks and the last group contains 4 disks. The striping size is 64 KB. The A3000 contains 64 MB of battery-backed NVRAM which can absorb some writes that may otherwise have gone to disk.

There are two reasons for investigating different systems. First, they allow us to draw conclusions about the effects of different hardware and software, e.g., drivers, main memory, and NVRAM. Second, having two systems serves as a check on our model; inaccuracies may show in one system but not the other. In addition, the RAID is closer in spirit to servers found in published SPEC data.

## 5.2 SPECsfs Characteristics

Just as we did with the Split-C/AM and the NPB, in this section we first characterize the SPECsfs benchmark which forms the workload for this chapter. We then describe the similarities and differences between this and previous work done to quantify NFS performance.

SPEC, while widely known for its CPU benchmarks, also produces an NFS benchmark, SPECsfs (formerly known as LADDIS) [107]. SPEC released the latest version, SFS 2.0, in 1997 [94]. Version 2.0 adds several enhancements. First is the addition of version 3 of the NFS protocol [82] as well as retaining NFS version 2. In addition, TCP can be used as a transport layer instead of UDP. The combination of these two variants results in four possible configurations (e.g., NFS version 3 on UDP and NFS version 2 on TCP). We focus on NFS version 2 running over UDP because this will comprise a large number of installed systems. Unless otherwise reported, all results are for systems running SFS 2.0, NFS version 2 using UDP. We do examine some TCP vs. UDP tradeoffs in

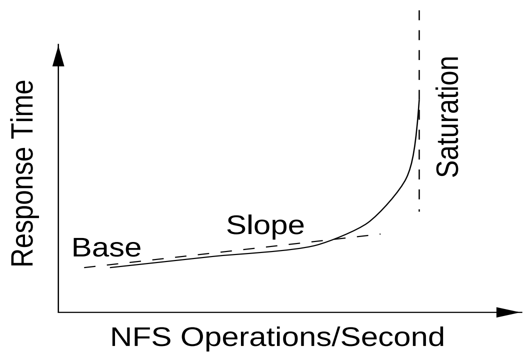


Figure 5.1: **Important Characteristics of the SFS Curve**
*This figure shows the important characteristics of all SFS curves: the* **base** *(i.e. minimum) response time, the* **slope***, which determines the rate of increase in response time as load increases for a linear region of the curve, and the* **saturation point** *at the peak operations sustainable.*

Section 5.5.2.

SPECsfs, as a synthetic benchmark, must define both the operation mix and scaling rules. The mix has been derived from much observation of production systems [94, 107]. SFS 2.0 uses a significantly different mix from SFS 1.0. With the decline of diskless workstations it was found that the percentage of writes had also steadily declined. Qualitatively, the SFS 2.0 operation mix is mostly small metadata operations and reads, followed by writes. The mix represents a challenge to latency tolerating techniques because of the small and synchronous nature of most operations. The two most common operations are `get_attributes` and `directory_lookup`; combined they make up 62% of the SFS 2.0 workload. Reads comprise 14% and writes comprise 7% of the operations with other metadata operations making up the remainder.

Learning from past benchmarking errors, SPECsfs also defines scaling rules for the data set. In order for a vendor to report a large number of operations per second, the server must also handle a large data-set. For every NFS op/sec, the clients create an aggregate of 10 MB of data. The amount of data accessed similarly increases; for each op/sec 1 MB of data is touched.

Unlike the SPEC CPU benchmarks which report point-values, SPECsfs reports a curve of response time vs. throughput. The reported response time is the weighted average of different operations' response times, where the weights are determined by the percentage of each operation in the mix. Figure 5.1 shows an abstract SFS results curve. The *signature* of the curve contains three key features: the *base response time*, the *slope* of the curve in the primary operating regime, and the *saturation point*.

At low throughput there will be an average *base* minimum response time. The base represents the best average response time obtainable from the system. The base will be determined by a host of factors, including the network, the speed of the CPU, the size of the file cache, the amount of Non-Volatile RAM (NVRAM), and the speed of the disks.

As load on the server increases, there will be a region where there is a linear relationship between throughput and response time. The *slope* signifies how well the server responds to increasing load; a low slope implies the clients cannot perceive a more loaded server, while a high slope implies noticeable delays as we add load. The slope will be affected by queuing effects in the network, at the server CPU, the server disks and the client CPU. However, a much more important role in the determination of the slope is the changing miss rate in the server file cache.

As the load further increases, a bottleneck in the system will limit the maximum throughput at the *saturation point*. The nature of the bottleneck will determine if the point is reached suddenly, resulting in a pronounced inflection, or gradually. Example bottlenecks include an insufficient number of clients, lack of network bandwidth, the speed and number of CPUs on the server, and an insufficient number of server disks.

SPECsfs is designed to isolate server performance. The benchmark therefore takes care to avoid client interactions to the extent they influence the accuracy of the load placed on the server. For example, differences in client file cache management make it difficult for the benchmark to control both the size and nature of the server load. The load generators thus call RPC procedures directly, bypassing the client filesystem and file cache. The results of this study therefore are primarily from the *server* perspective. Average response time results on production systems will depend on the type of workload (e.g. attribute vs. bandwidth intensive) and size of the client caches. See [44] for an examination of the effects of client caches on server load.

Using SPECsfs as a workload is quite novel in an academic setting. One factor limiting previous work was the scale on which the benchmark must be run in order to obtain meaningful results. For example, an insufficient number of disks would limit our understanding of CPU bottlenecks. Also, industry reported results are for very large systems, often beyond the range of a dedicated academic testbed. For example, at the completion of this study, one of our testbeds immediately went into production use.

From our “grey-box” perspective, SPECsfs is quite useful because it fixes all the NFS parametric inputs but one: the server load. We can thus restrict the parameter space primarily to the network. At the same time, our choice of SPECsfs clearly limits our understanding of NFS in several ways; it is attribute intensive and it does not model the client well. Practically, however, our choice allows us to interpret industrial data published on SPEC’s website in the framework presented in this

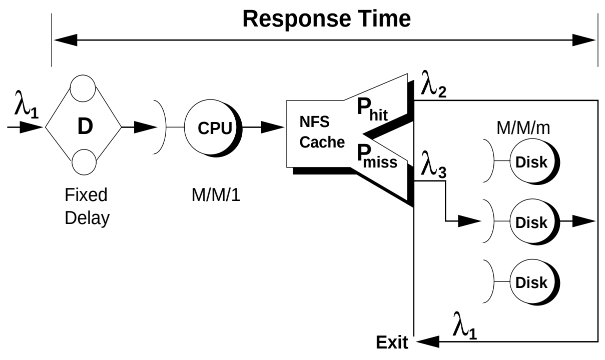


Figure 5.2: **SPECsfs Analytic Model**
*This figure shows the simple analytic model used to validate the NFS results. The model assumes a poisson arrival rate of $\lambda_1$ requests per second. The model then uses a fixed delay center to model the network, an M/M/1 queue to model to CPU, a splitter to captures NFS cache effects, and finally an M/M/m queue to model the disk subsystem. The parameters for each component were determined by empirical measurement.*

paper. We perform a brief analysis of industrial data in Section 5.6.

## 5.3 SPECsfs Analytic Model

In this section we build a simple analytic model of the entire NFS system, focused on the server portion of the system. The goal of the model is to provide a framework for understanding the effects of changes in $L$, $o$ and $G$ on the SFS results curve. We then compare the predictions of the model against two measured SFS curves. If the model and experimental data agree, we can have reasonable confidence in both. Significant differences between the two would show where either the model fails to describe the system, or where the system is mis-configured and thus not operating "correctly". In either case, more investigation may be needed to resolve the discrepancy. We conclude the section with predictions on the sensitivity of the system to network parameters.

### 5.3.1 Model Construction

Figure 5.2 shows the queuing network we use to model an NFS server, adopting the simple techniques described in [57, 65]. The model consists of a CPU, disks, NFS cache, and a delay center. Our model ignores queuing delays in the controllers and I/O bus; they can easily support the load placed on them given the small nature of most requests.

We assume that the arrival rate follows a Poisson process with a rate of $\lambda_1$ requests per second. Because the departure rate must equal the arrival rate, the departure rate is also $\lambda_1$.

The CPU is the simplest component of the system. We model it as an M/M/1 queue. We derived the average service time, including all sub-systems (e.g. TCP/IP protocol stacks, and the local filesystem, UFS) from experimental measurement. For the SCSI based system, the measured average service time was 900 $\mu$s per operation. The RAID system has a lower average service time of 650 $\mu$s. We provide a detailed investigation the components of the service time in Section 5.5.3.

Most NFS operations have the potential to be satisfied by an in-memory cache. Only 7% of the SFS 2.0 mix are writes and these must bypass the cache—NFS version 2 semantics require that they exist in stable storage before the write completes. The 64 MB of NVRAM in the RAID can cache writes, however. The file cache size, and corresponding miss rate, are critical to determining the base response time as well as the slope of the SFS curve. However, the SFS strategy of increasing the data set size per op/sec places an upper limit on the effectiveness of a cache.

We model the NFS caches (both in-memory and NVRAM) as a splitter. The probability of a hit is given as $P_{hit}$ and of a miss as $P_{miss} = 1 - P_{hit}$. On a hit, the request is satisfied and leaves the system. Because the data set accessed by SFS increases with the load, $P_{hit}$ is a function of $\lambda_1$. We use a simple approach to computing $P_{hit}$. We take the main memory size plus the NVRAM size and divide it by the accessed data set size. In terms of our model, the splitting a Poisson stream results in two Poisson streams, $\lambda_2$ and $\lambda_3$. The rate of requests going to the disks is easily derived as $\lambda_3 = P_{miss}\lambda_1$

The disks are modeled by an M/M/m queue where m is equal to the number of disks. We have empirically observed using the `iostat` command an unloaded average service time of 12 ms for the IBM drives. We use the same value to model the Seagate drives.

We fold the remaining components into a fixed delay center with a delay of $D$. These components include the overhead in the client operating system, fixed costs in the server, and the network latency. The use of fixed delay greatly simplifies the model, allowing us to focus on the important elements of the server. We can still obtain reasonable accuracy using a fixed delay center, however. We empirically observed a $D$ of 3.4 msec. This fixed delay parameter was obtained by observing a small request rate of 300 op/sec on the RAID system. At that rate, the entire workload fits into memory, so nearly all disk requests have been eliminated.

| | SCSI | | RAID | |
|---|---|---|---|---|
| Range(op/sec) | 200-1050 | | 500-1400 | |
| | Q-model | Measured | Q-model | Measured |
| Slope $\mu$sec per op/s | 8.5 | 14.3 | 10.4 | 18.9 |
| Y-intercept | 8.0 | 4.52 | -6.8 | -0.04 |
| Base | 8.6 | 7.3 | 4.3 | 4.5 |
| $r^2$ | 0.93 | 0.99 | 0.98 | 0.96 |

Table 5.1: **SPECsfs Linear Regression Models & Accuracy**
*This table demonstrates linear regressions of the SFS queuing-theoretic models and measured data. The table shows the slope of the SFS curve, (increase in response time vs load), the Y-intercept, the base performance at 200 and 500 ops/sec, and the coefficient of determination ($r^2$).*

### 5.3.2 Model Accuracy

Figure 5.3 shows the accuracy of our simple queuing model compared to the measured data for our baseline systems. The baseline systems have the minimum $L$, $o$, and $G$, and thus maximum performance in all dimensions. In order to measure the slope of the SFS curves, we performed a linear regression on a range of measured data (200-1050 for the SCSI and 500-1400 for the RAID). Table 5.1 shows that within these ranges a linear model is quite accurate; the $r^2$ values are 0.99 (SCSI) and 0.96 (RAID).

At a qualitative level, we can see that the NFS cache sizes have a significant impact on the shapes of both the measured and modeled systems. Below 500 ops/sec for the RAID, the SFS curve is fairly flat because the cache is absorbing most of the requests. The SCSI system, with its small cache, has a continuously rising curve. The slope of the RAID is much steeper than the SCSI system for exactly the same reason—differences in cache size.

At a more quantitative level, across the entire range of throughputs the relative error of the queuing model is at worst 24% for the SCSI and 30% for the RAID. This is reasonably accurate considering the simplicity of the model, e.g., we do not model writes by-passing the file cache. Unfortunately, the queuing model consistently under predicts the slopes of the SFS curve. Linear regressions of the queuing model predict slopes of 8.5 (SCSI) and 10.4 (RAID) $\mu$s per op/sec. These are substantially lower than the 14.3 and 18.9 $\mu$s per op/sec for the measured slopes.

The shape of the inflection point is a second inaccuracy of the model. In the SCSI system, the measured inflection point is quite muted compared to the modeled curve. The last point of the modeled SCSI curve, which has no measured counterpart, shows a rapid rise in response time in the

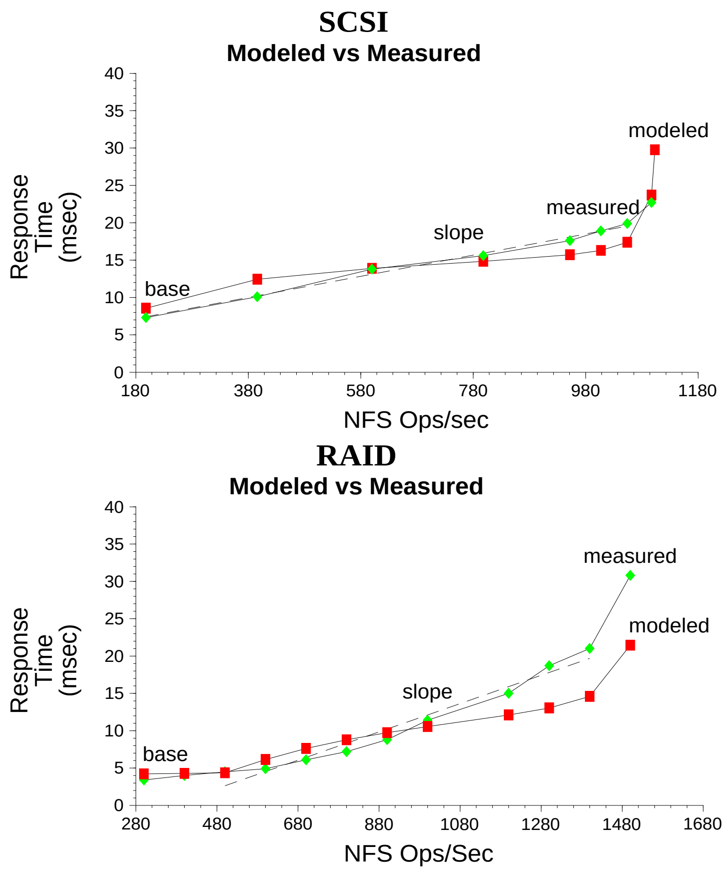


Figure 5.3: **SPECsfs Modeled vs. Measured Baseline Performance**
*This figure plots the modeled as well as baseline SFS curves for the SCSI system (top) as well as for the RAID based system (bottom).*

99+% utilization regime. The real system, however, will not enter into that regime. We explore the effects of high utilization in Section 5.5.3.

In spite of the inaccurate slopes and inflection point, the queuing model is quite accurate for most of the operating regime. Only at very high utilization does it deviate much from the measured values. Interestingly, the $r^2$ values in Table 5.1 show that the live system behaves in a linear fashion across almost all of the operating regime, more so than the model would predict.

For the purposes of capacity planning, the queuing model may be quite acceptable because operating at the extremes of the performance ranges is undesirable. A lightly loaded system wastes resources, while a system operating near capacity results in unacceptable response times.

### 5.3.3 Expected Sensitivity

The model provides a concise conceptual framework; we use it to predict the impact of changing each LogGP parameter. The delay center captures the network latency term. We thus model an increase in $L$ as a linear increase in $D$, thereby changing the base response time. Each $\mu$s of added $L$ should add 2 $\mu$s to the base response time, because each operation is a synchronous request-response pair. An increase in $L$ should have no effect on the slope or saturation point. In the next section, we see that our model predictions for the slope and saturation point are accurate for a wide range of $L$ values, but not for extreme ranges. We also see that the model consistently under-predicts the sensitivity of the base response time to $L$.

Increasing $o$ we expect changes to all three components of the SFS signature. The response time should increase because of the client overhead encapsulated in the $D$ parameter and increased service time on the CPU. The slope should increase due to queuing delays at the CPU. The most important effect of $o$ is that the saturation point may be reached sooner. Because of the increased service time on the CPU, it will reach maximum utilization sooner. If, however, some other component of the system were the bottleneck, we may observe a region where the saturation point is insensitive to $o$. For our two servers, the model predicts that the CPU will be the bottleneck. We model the relationship between the saturation point and overhead as:

$$Saturation = \frac{1}{Serv + 2.4o}$$

where $Serv$ is the average CPU service time per operation previous measured in Section 5.3.1. The coefficient of 2.4 is the average number of messages per NFS operation. We model 2 messages per operation: a request and a reply. However, as $o$ is incurred on every message, we also model 2 extra fragments per read or write due to MTU effects. Given the frequency and size of reads and writes, the MTU effects raise the constant to 2.4. The next Section will show our model of sensitivity to $o$ to be quite accurate.

The bandwidth, $\frac{1}{G}$, is not captured well by any single parameter of the model. If we assume that requests are uniformly distributed in time, $G$ will have no effect until $\lambda_1 > \frac{1}{G}$. Indeed, this is a good test to see if requests are bursty or not. If requests are bursty then we expect that the NFS system would be quite sensitive to changes in $G$.

## 5.4 Previous Work on NFS Performance

Due to its ubiquity as a distributed filesystem there is vast body of work on NFS. Fortunately, [82] contains an excellent bibliography and summary. This section does not try to document all previous NFS work; rather we categorize related work and introduce papers which describe previous results upon which our work builds.

NFS studies fall into roughly three categories: protocol changes, client/server enhancements, and performance evaluation. Although papers contain some element of all three, often they focus in a single area. Many NFS protocol studies explore changes to improve filesystem semantics [68, 71, 77, 83], however, a few are performance oriented [38]. Enhancement studies focus on client/server design changes rather than protocol changes [37, 100]. For example, [59] looks at write gathering to improve performance, and a comparison of TCP vs. UDP and copy reduction techniques are examined in [70].

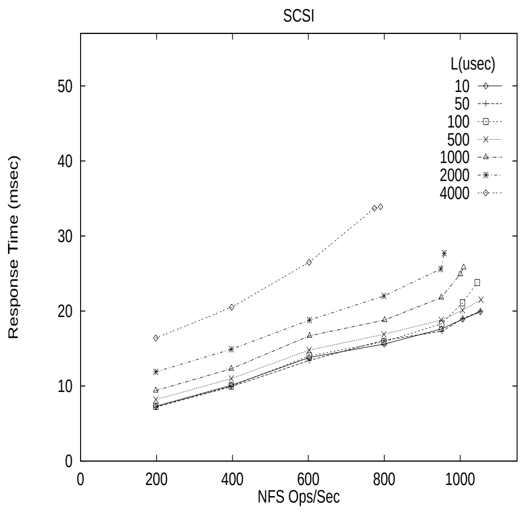
Our work clearly falls into the performance analysis category, using network performance as the dependent variable. The work in [96] takes an interesting perspective compared with ours. Instead of examining NFS performance as a function of the network, it examines network performance as a function of NFS load. It found that modest NFS load can severely degrade Ethernet performance. With the advent of multi-gigabit switched LANs, network loading due to NFS traffic is a minor problem compared with the days of shared 10 Mb Ethernet.

Although it deals with differences between NFS version 2 and version 3 [82], performs much performance analysis to justify the changes. It found that a server running NFS version 3 is roughly comparable to the same server running version 2 with NVRAM. However, little exploration of the impact of the version 3 protocol changes is made with relation to network performance.

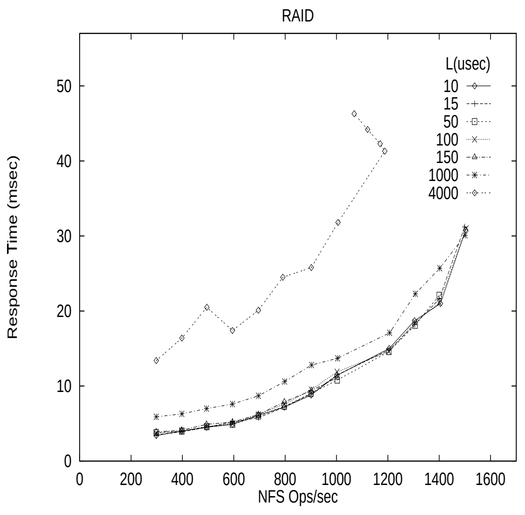
An extensive bottleneck analysis is presented in [108]. The book examines the peak throughputs of real-world components (e.g. CPU, disks, network) and characterizes where the saturation point will be for different configurations. An interesting result of the work is that most servers in the SPEC results are over-provisioned with disks.

Section 2.6.2 showed that perhaps the closest study in spirit to the experiments in this chapter is [21]. That work was primarily concerned with NFS performance over congested ATM networks. They found that a high $L$, in the 10's of milliseconds, was quite detrimental. A trace-fed simulation was used rather than a live system. Moreover, their custom workloads make a quantitative comparison to our work difficult.

We examined the methodology of [49] in Section 2.6.4. Recall that their conclusions are



(a)



(b)

Figure 5.4: **SPECsfs Sensitivity to Latency**
*This figure plots the SFS curves as a function of latency in microseconds. Measurements for the graph on the left were taken on the SCSI system, while measurements for the graph on the right were taken on the RAID system.*

quite similar to ours: CPU overhead is a dominant factor in NFS performance. We compare their results to ours in greater detail in Section 5.6. Some of their most interesting data, however, came from their direct measurement of a large production system.

## 5.5 Sensitivity Results

Given our methodology and NFS characterization, we now quantify the effect of varying LogGP parameters. As in previous chapters, we independently vary each of the parameters in turn and observe the resulting SFS curves. For each parameter, we attempt to explain any observed changes to the SFS signatures based on the model developed in Section 5.3. In addition, we measure the most appropriate sensitivity for each parameter. For latency this is change in base response time as a function of $L$. For overhead, it is the change in saturation point as a function of $o$.

### 5.5.1 Latency

Historically end-to-end latency is often thought of as the critical parameter for NFS performance [21, 80]. In terms of the LogGP model, the typical definition of latency includes **both** $L$ and

$o$. In this section, we examine solely the $L$ term. By focusing on latency alone, we can better quantify the effects of the network itself, rather than mixing the effects of the network and end-system.

**Response to Latency**

Figure 5.4(a) shows a range of SFS curves resulting from increasing $L$ for the SCSI system. Likewise, Figure 5.4(b) shows the results for the RAID. The range of $L$ has been scaled up from a baseline of 10 $\mu$s to 4 msec. For comparison, most LAN switches have latencies in the 10's of $\mu$s. Most IP routers, which would be used to form a campus-wide network, have latencies of about a millisecond. Thus the range explored in Figure 5.4 is most likely what one might find in an actual NFS network. We will explore the effect of very high WAN-class latencies in Section 5.5.2.

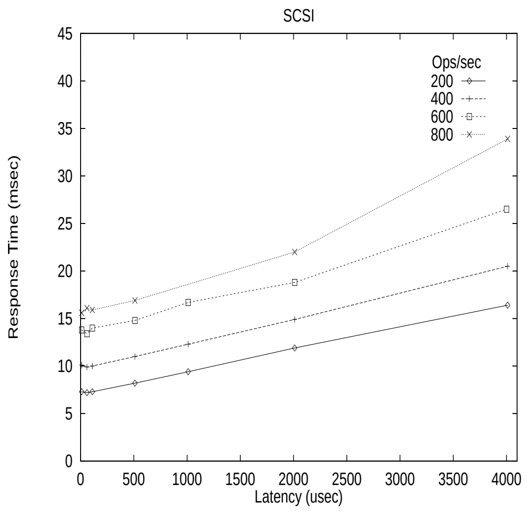
We have truncated the SCSI curves at the saturation point, to increase readability, but present the full RAID data. Figure 5.4(b) shows the 4 msec RAID curve "doubling back". Because the SFS benchmark reports the response time vs. delivered throughput, as opposed to offered load, attempts to exceed the saturation point can result in *fewer* operations per second than attempted. We will explore this effect in greater detail in the discussion of overhead.

As predicted by the queuing model, the measured data shows the primary effect of increased $L$ is to raise the base response time. Also, as predicted by the model, the slope does not change. Modest changes in $L$ do not affect the saturation point. However, a high $L$ can cause the saturation point to fall, as shown by both the 4 msec curves. The reason for the drop is that insufficient parallelism exists due to lack of client processes. We have tested this hypothesis by increasing the number of load generator processes on the client. An unusual side effect of increasing the number of load generators is a change in the slope of the SFS curve. We therefore use the minimum number of load generators that can saturate the system in the baseline case even if it results in lower saturation points as we scale $L$.

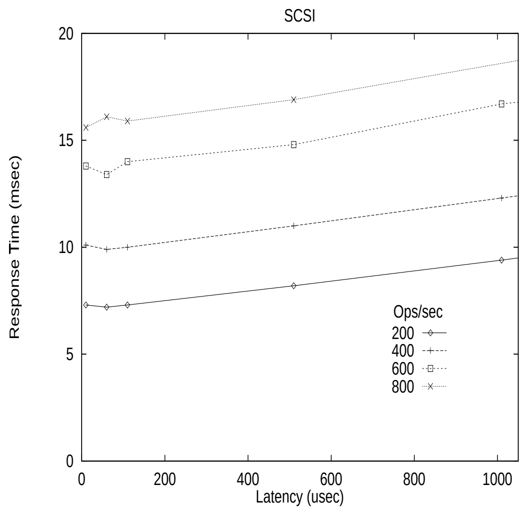
Returning to the base response time, a key question is what the rate of the increase to $L$ is. That is, for each $\mu$s of $L$ added, what is the corresponding increase in response time? The next section explores this question in greater detail.

**Sensitivity to Latency**

Figure 5.5 shows the sensitivity of response time as a function of $L$ for a range of throughputs, i.e., each line is a vertical slice though Figure 5.4. Two distinct sensitivity regions are observable. Figure 5.5(a) shows the first region has a constant sensitivity of 3 $\mu$s of response time for each



(a)



(b)

Figure 5.5: **SPECsfs Latency vs. Response Time**
*This figure plots response time as a function of latency in microseconds. Measurements were taken on the SCSI system. The graph on the left shows a range of $L$ up to 4000 $\mu$s. The graph on the right shows that up to 150 $\mu$s there is little sensitivity to $L$.*

$\mu$s of added $L$ between 150 - 4000 $\mu$s. This is quite a bit higher than the 2 predicted by the model in Section 5.3.2. Figure 5.5(b) shows a completely insensitive region between 10 and 150 $\mu$s.

An important result is that in the sensitive region, all the sensitivity curves are a constant 3 across an order magnitude change in $L$. Given that the system is responding in a linear fashion, there may be an accurate way to model it. However, the constant of 3 is quite a bit higher than the constant 2 predicted by our simple model. A more complex model is needed to account for the discrepancy.

In the insensitive region we can see there is little, if any, measurable change in response time as we vary $L$. For the same range of $L$, the same results applies to the RAID as well. This has important implications for switch and interface designers as these operate in the 10's of $\mu$s region. From an NFS perspective, a LAN switch adding 10 $\mu$s of delay per hop would be quite acceptable.

### 5.5.2 High Latency

The original NFS protocol was not designed to operate in environments with very high $L$. NFS version 3 added several latency tolerating techniques, most notably asynchronous writes [82]. In this section, we examine the effects of very high $L$, in the 10's of millisecond range. For example, WANs typically have an $L$ ranging from 10's to 100's of milliseconds.

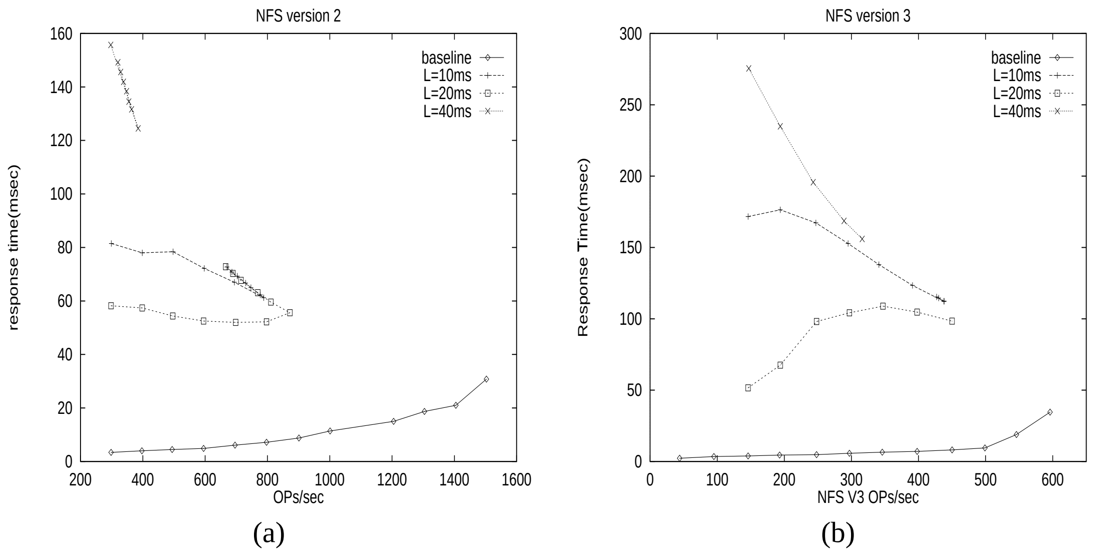
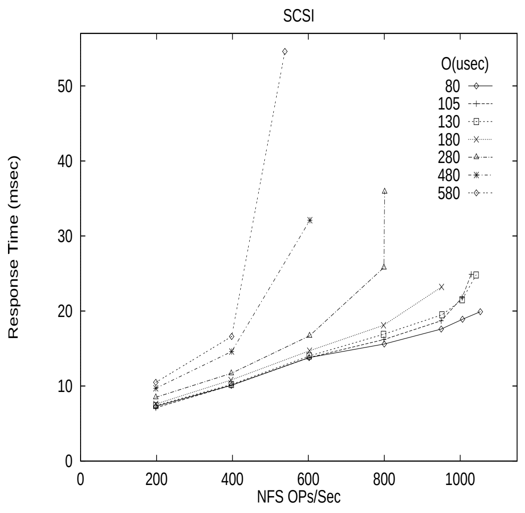


Figure 5.6: **Effects of Very Long Latency**
*This figure plots the SFS curves as a function of very high latencies on the RAID. Measurements for the graph on the left were taken on for NFS Version 2 over UDP, while measurements for the graph on the right were taken using NFS Version 3 running over TCP. The figure is designed to show the relative performance degradation for each version as neither the operations/sec or the response times between versions is comparable.*

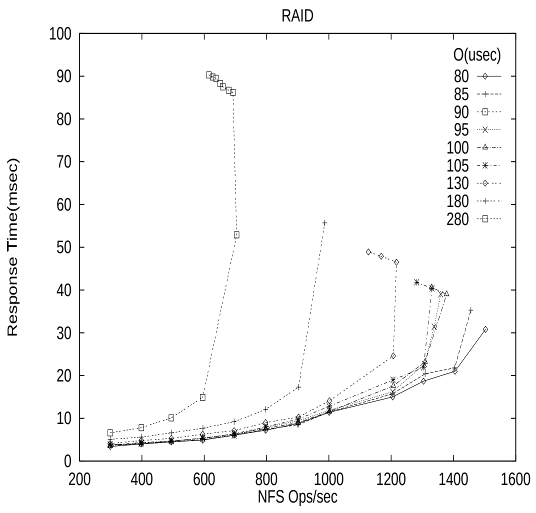
Figure 5.6 compares the relative effectiveness of NFS version 2 running on UDP, a typical configuration, to version 3 running on TCP for networks with high $L$. The experiment varies both the NFS version and network transport at once to better understand the total impact of an upgrade. Typically, operating systems that ship with version 3 also allow TCP as a transport layer. Both the throughput and response times between NFS version 2 and version 3 are not comparable; thus we examine the percentage of performance loss as we scale $L$.

Figure 5.6(a) shows, as expected, that the “classic” NFS V2/UDP performance over WAN latencies is dismal. First, the base response time is hyper-sensitivity to high latency in that it is much greater than one would predict from the simple model. Second, very little of the peak load is obtainable. NFS Version 3 over TCP is able to handle long latencies much better than version 2 over UDP. Figure 5.6(b) shows that even at an $L$ of 40 msec (a round trip of 80 msec), Version 3/TCP can sustain 50% of the peak throughput without adding extra clients.

A notable effect on both versions is that average response time *decreases* as the load increases. This could be caused by a number of effects. One possible effect could be the interaction of a light workload with the RPC and TCP transport protocols. These algorithms constantly probe



(a)



(b)

Figure 5.7: **SPECsfs Sensitivity to Overhead**
*This figure plots the SFS curves as a function of overhead in microseconds. Measurements for the graph on the left were taken on the SCSI system, while measurements for the graph on the right were taken on the RAID system.*

the network looking for more bandwidth. Under a light workload however, an insufficient number of packets may be sent for the protocol to reach a stabilization point in its time-out/re-try algorithm. As both curves are consistent with this theory, it begs the question as to the performance of these algorithms [56] under a very light load.

### 5.5.3 Overhead

Software overhead, the orphan of networking analysis, permeates the design space because it affects all aspects of the SFS signature curve. We focus our analysis efforts, however, on its effect on the saturation point. Not only are these effects likely to be the most pronounced, but they also will greatly impact the machine size needed to sustain high loads.

#### Response to Overhead

Figure 5.7 shows the SPECsfs curves for both the SCSI and RAID systems while scaling overhead from a baseline of 80 $\mu$s. For the SCSI system we have truncated the results at the saturation point to make the graph more readable.

Figure 5.7 shows that the base response time increases as we scale $o$, and the measured results are close to the model predictions. The slope of the SFS curve is fairly insensitive to $o$ until throughput is near the saturation point. A queuing model gives us nearly the same result; the slope of the SFS curve will not change drastically with respect to $o$. The most dramatic effect of $o$, however, is on the saturation point.

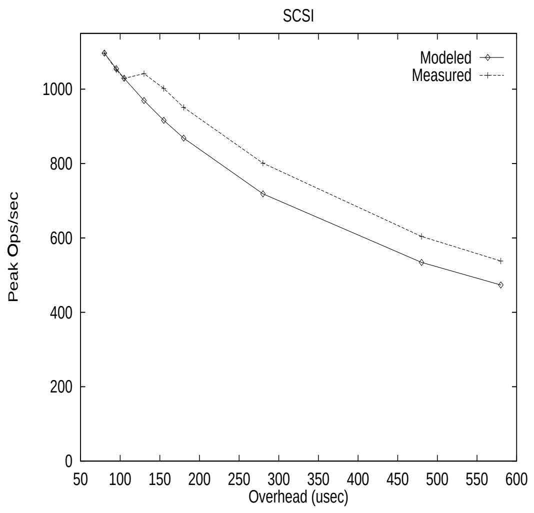
Figure 5.7(b) shows what happens to the saturation point in the RAID when system capacity is exceeded; both response time and throughput degrade slightly as offered load exceeds the saturation point. A less dramatic version of this effect was observable as we scaled $L$ as well. An interesting open question is how well the system responds to these extreme conditions, i.e., how much performance is obtainable when the offered load is 150% of peak? Queuing theoretic models tell us that response time should increase to infinity as offered load nears 100%. Figure 5.7(b) shows that in a real system (which is a closed system) the response time hovers around an overhead-dependent maximum while the delivered throughput slowly decreases. Feedback loops built into the RPC layer, based on algorithms in [56], keep the system out of the realm of very high response times, instead forcing the entire system towards lower throughputs. The algorithms are quite effective; rather than a complete system breakdown we observe small degradations in throughput and response time. A full investigation of these effects, however, is beyond the scope of this work.

Because we are scaling a processor resource, the lower saturation point must be due to the higher service time of the CPU. In the next sections we will explore the nature of the saturation point. We first derive the sensitivity curve and then examine the components of the service time for both the SCSI and RAID systems.

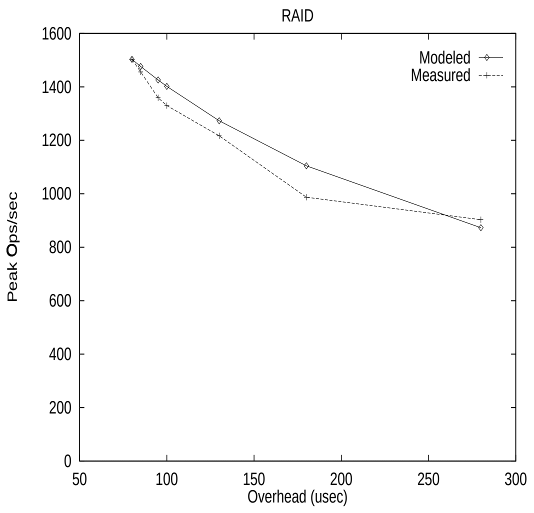
### Sensitivity to Overhead

Figure 5.8 shows the relationship between overhead and throughput. The modeled line shows where the CPU reaches 100% utilization in the model presented in Section 5.3, while the measured line is derived from the results in Figure 5.7. The most interesting aspect of both systems is that the peak performance drops immediately as we add overhead; unlike the response to latency, there is no insensitive region. Therefore, we can easily conclude that the CPU is the bottleneck in the baseline system. Also for both curves, the response to overhead is non-linear, i.e., for each $\mu$s of added overhead, the peak drops off quickly and then tapers out.

To determine the accuracy of the model, we performed a curvilinear regression against the overhead model in Section 5.3.2. The $r^2$ values of .99 for the SCSI and .93 for the RAID show that our model is fairly accurate. The sensitivity to $o$ agrees well with the model.



(a)



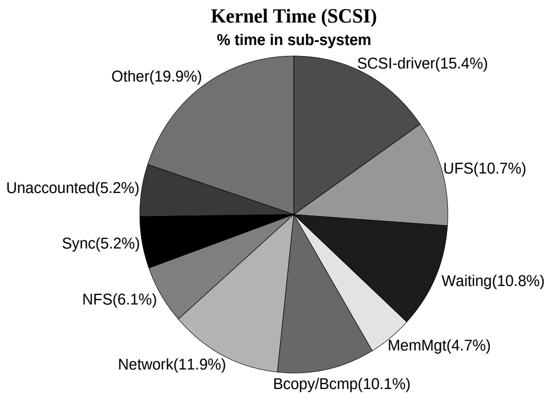
(b)

Figure 5.8: **Peak Throughput vs. Overhead**
*This figure plots the saturation point as a function of overhead in microseconds. Measurements for the graph on the left were taken on the SCSI system, while measurements for the graph on the right were taken on the RAID system.*

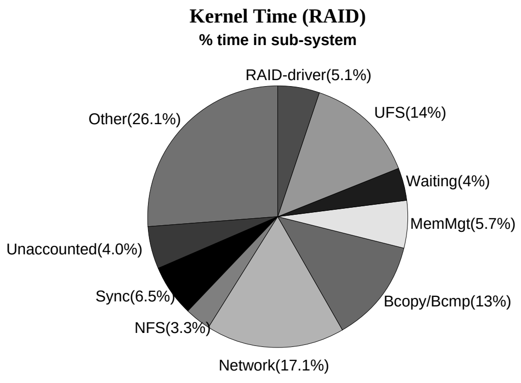
## Examining Overhead

The SCSI and RAID system both use the same CPU, operating system, network, and nearly the same number of disks. Yet RAID's saturation point is much higher. An obvious question is the reason for the lower performance of the SCSI system. Figure 5.9 reports the percentage breakdown of different components of CPU near the saturation point for both the SCSI and RAID. Because the monitor itself (`kgmon`) uses some CPU, it is not possible to actually reach the saturation point while monitoring the system. The relative areas of the charts show the difference in average time per operation, including CPU idle time, of 1 msec for the SCSI and 714 $\mu$s for the RAID.

The most obvious difference is the time spent in the device drivers; the FAS SCSI drivers spend an average of 150 $\mu$s per NFS operation while the RAID drivers spend an average of only 36 $\mu$s. The networking stacks and filesystem code comprise the two largest components of the CPU time. However, an interesting feature of both systems is that a significant amount of the service time (20% and 26%) is spent in general kernel procedures which do not fall into any specific category. There are a myriad of these small routines in the kernel code. Getting an order of magnitude reduction in the service time would require reducing the time of many sub-systems. Much as was found in [26, 60] there is no single system accounting for an overwhelming fraction of the service time.



(a)



(b)

Figure 5.9: **Time Breakdown Near Peak Op/sec**
*These charts show the percentage of time spent in each sub-system when operating near the saturation point. Measurements for the graph on the left were taken on the SCSI system at 1000 ops/sec, while measurements for the graph on the right were taken on the RAID system at 1400 ops/sec. The area of each chart shows the relative time per operation, including idle time (i.e., waiting on I/O), of 1 msec for the SCSI and 714 $\mu$s for the RAID.*

### 5.5.4 Bulk Gap

We choose to examine sensitivity to bulk Gap, $G$, as opposed to the per-message rate $g$. First, networking vendors often tout per-byte bandwidth as the most important metric in comparing networks. Using our apparatus we can quantify its sensitivity (and thus importance). Secondly, for the SPECsfs benchmark, $g$ is quite low (in the 1000's msg/sec range) and is easily handled by most networks.

Unlike overhead, which is incurred on every message, sensitivity to Gap is incurred only if the data rate exceeds the Gap. Only if the processor sends data in an interval smaller than that specified by $G$ will it stall. The clients and server could potentially ignore $G$ entirely. Recall the burst vs. uniform models for gap presented in Section 3.2.2. At one extreme, if all data is sent at a uniform rate that is less than $G$ we will not observe any sensitivity to Gap. At the other extreme, if all data is sent in bursts then we would observe maximum sensitivity to Gap.

Because SPECsfs sends messages at a controlled rate, we would expect that message intervals are not bursty and the benchmark should be quite insensitive to Gap. Figure 5.10 shows that this is indeed the case. Only when the bandwidth ($\frac{1}{G}$) falls from a baseline of 26 MBs to a mere 2.5 MB/s do we observe any sensitivity to $G$. We are thus assured that the SFS benchmark is not bursty.

Measured production environments, however, are quite bursty [49, 66]. Our measured sen-

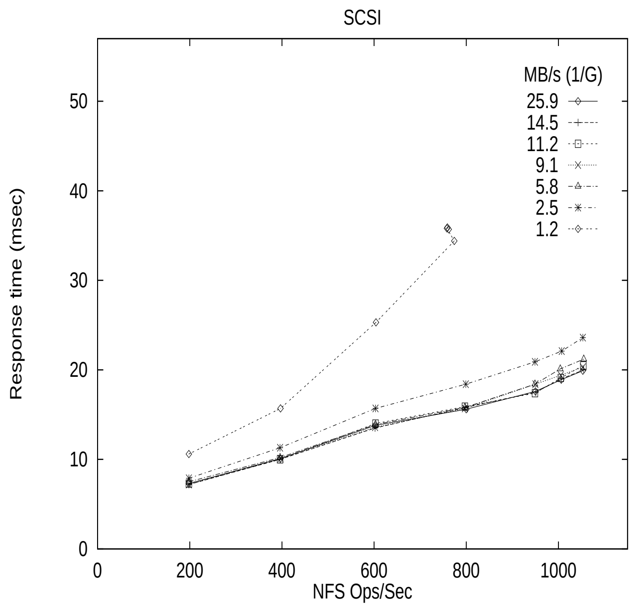


Figure 5.10: **Sensitivity to Gap**
*This figure plots the SFS curves as a function of Gap in microseconds. Measurements for the graph were taken on the SCSI system.*

sitivity to $G$ is thus lower than what one might expect in a production environment. We explore the implications of bursty networks to the sensitivity of $G$ in more detail in Chapter 5.6.

Figure 5.10 shows queuing delays at very low bandwidths that are not captured by the model. There is a slight increase in slope at a bandwidth of 2.5 MB, and at 10 Mb Ethernet speeds (1.2 MB/s) there is a noticeable increase in slope. Replacing the simple delay center with a more sophisticated queuing network would capture these effects. However, given the low bandwidths at which these effects occur, we have made a quite reasonable tradeoff between model simplicity and accuracy.

## 5.6 NFS Summary

In this section, we describe the implications of our sensitivity results. As in previous chapters, we organize our results around the four areas of performance analysis, application behavior, architecture and modeling.

### 5.6.1 Performance Analysis

This chapter demonstrates that the TCP/IP apparatus is quite effective at adding controllable delays to various system components. One apparent drawback of our apparatus is that we could not reduce overhead to low levels. It would thus appear that our apparatus is limited in its ability to answer questions in the regime of low overhead. However, our methodology can overcome this limitation. Rather than answer the question "what is the benefit of reduced overhead?" by measuring the effects of low overhead directly, our method and apparatus allow indirect observation. The shape of the sensitivity curve will quantify the effects of reducing overhead in a real system, limited by Amdahl's Law of course. A steep slowdown curve would show that overhead reduction would have an immediate, positive impact.

A new use of our apparatus in this chapter was the scaling of latency into the WAN range. While performing adequately in this context, it was clear from the results that neither NFS nor the SPECsfs benchmark are suited to run over this class of networks. While not entirely surprising, our methodology showed it can quantify some of the strange effects that happen at these long latencies.

### 5.6.2 NFS Behavior

On the behavior side, one primary result is that in typical SAN and switched LAN environments, the latency is quite satisfactory for NFS. Latencies under 150 $\mu$s are easily obtainable in these environments, even when cascading switches. Further reductions will have little benefit.

In the WAN range, we have seen that the changes to NFS version 3 indeed improve performance. However, such latencies are still a significant performance drag, as was also found in [21]. Qualitatively, the changes in Version 3 have raised the level of NFS performance over WANs from "unusable" to merely "slow". Even with these enhancements however, it may not be economically viable to use NFS over WAN ranges. Given the low cost of disk storage compared with WAN links, it may make more sense to replicate the entire data-set. Even for large amounts of data, the storage requirements are cheap compared with the recurring costs of WAN links.

The SPECsfs workload has minimal bandwidth needs and is quite regular; generating traffic on the order of single MB/s. However, real networks exhibit quite bursty behavior and thus bandwidth requirements would be higher, but not into the gigabit range.

### 5.6.3 Architecture

In the architectural arena, overhead continues to be a performance limiter, much as it is for the Split-C/AM programs, and this is where significant performance improvements could be made. A similar conclusion as was found in [49] as well. The study examined three points in the networking space (ATM, Autonet and FDDI), rather than systematically varying overhead. However, it is encouraging that two different studies have come to the same conclusions by much different methods.

Although the networking overhead was only 20% of the entire service time, that does not mean that attempts to reduce $o$ will yield marginal results. Indeed, networking overhead is one of the primary components of the service time. However, a number of subsystems must be improved at once for significant progress to be made. For example, a combination of novel overhead reducing interfaces between major OS sub-systems, disks and network interfaces might yield significant improvements.

Turning to latency, the architectural improvements in $L$ from IP switches, which place much of routing logic in silicon, will have a large impact on NFS. The order of magnitude drop in $L$ from the millisecond to the 10-20 $\mu$s region [98, 99] will expand the range of NFS to a much wider area. Recent switches also offer increased port densities, ranging to 100's of ports at 100 Mb Ethernet speeds. A network composed of these low-latency, high-density IP switches would expand the range of NFS service to a whole campus, even multiple campuses, instead of it's traditional domain, a building. The implications of such an expansion are interesting; NFS service could reach a much larger number of machines than previously possible.

For bandwidth, network technologies such as switched 100Mb Ethernet and 155 Mb ATM provide plenty of bandwidth for NFS workloads. Given that most NFS packets are quite small, overhead, rather than bandwidth or latency, will still be the dominant factor facing future network designers.

### 5.6.4 Modeling

Simple queuing models are quite effective in analyzing the behavior of an NFS server. We were able to model changes in response time, slope and saturation point for a variety of parameters, although more investigation is needed to better describe the effect of latency on response time. We empirically measured and validated the inputs to the model. However, one could obtain close to the same inputs for a specific configuration by looking at the data published data on SPEC's website [95].

Using the SPEC data and the results of this study, it is relatively straightforward to deduce the parameters of the queuing model for a specific configuration from the published SFS curves. A word of caution is needed when using this approach: mixing parameters for different hardware/software configurations, particularly overhead, can be quite inaccurate.

Deconstructing the NetApp F630 and AlphaServer 4000 5/466 using the data from the SPEC webpages is an instructive exercise. They both have roughly the same CPU (500 MHz Alpha), but the Alphaserver has twice the main memory and disks as the NetApp box. The NetApp box however, has half the base response time, a much lower slope, and a higher saturation point. Putting the results into the context of this work, we can conclude that Network Appliance was quite successful in their bid to reduce overhead via a specialized operating system [51]. Another approach to obtaining a higher saturation point is to add processors, demonstrated by the 18 CPU Sun system. Such an approach would not reduce the base response time, however, unless the operating system can parallelize a single NFS operation.

# Chapter 6

# Investigating Overhead Reduction

*There is no such thing as a failed experiment, only more data.* —Max Headroom.

The previous three chapters have shown that of all the LogGP parameters, most applications exhibit considerable sensitivity to overhead. That results points to overhead reduction as a promising avenue for improving application performance. In this chapter, we present preliminary work on a novel software architecture, called SPINE [40], which was constructed with overhead reduction as a specific design goal. SPINE allows the application developer to reduce overhead by partitioning the application between the host CPU and network interface. The potential advantage of the SPINE approach is that the network interface may be able to reduce overall data movement and control transfers, both of which impact $o$, at a cost of an inflated gap and latency. The key to this overhead-reduction technique is to limit the inflation of the other parameters.

There are many potential ways which to reduce overhead. Fortunately, they can be classified into three general methods:

- **Restructure the application.** In this approach, the application is changed to reduce and/or aggregate communication. An example of this approach can be seen in the successive versions of the EM3D application presented in [28]. The simplest versions are quite sensitive to $o$, but a series of progressively more complex transformations alters the application until it is primarily sensitive to $G$.

- **Change the communication protocol.** A straightforward method of reducing overhead is to use a faster communications layer. For example, NFS was initially built on UDP instead of TCP for exactly this reason [70]. The tradeoff is that the application may have to re-implement functionality in the higher overhead layer in order to use the other one.

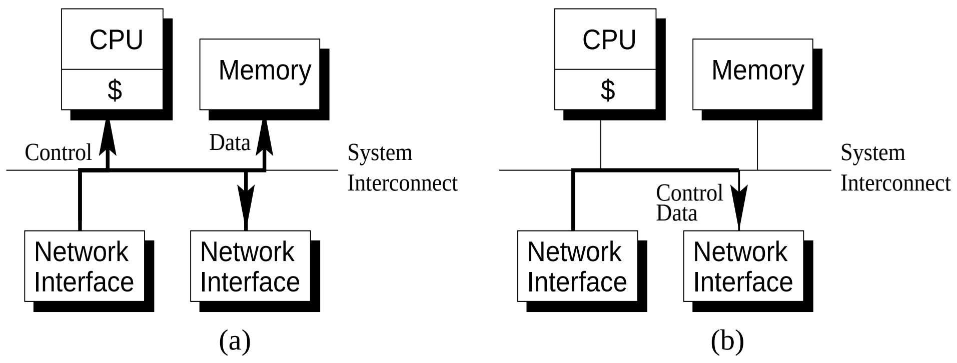


Figure 6.1: **SPINE Approach to Overhead Reduction**
*This figure shows the basic overhead reduction technique used in SPINE. In a normal system, shown in figure (a), the CPU must handle control and/or data from the network interface. Control and data flow for messages in SPINE, as shown in figure (b), can avoid the main CPU entirely. A unique aspect of SPINE is the ability to safely run arbitrary application code on the network processor.*

- **Add functional units.** A familiar approach in the architecture community, this approach has been spurned in the network community in recent years. The basic idea is to partition the problem such that multiple hardware units can pipeline packet processing. As we reduce $o$, we hope that the additional cost in terms of $L$,$g$ and $G$ will not be too high—or may even be less. A DMA engine is a well known example of an added functional unit that reduces $o$ and often greatly improves $G$. Although exploiting parallelism in communication has been explored in the context of Symmetric Multiprocessors [91], there has been surprisingly little work in more specialized support.

In this section, we will explore a combination of restructuring and adding functional units to reduce overhead in an IP router. In the terminology of this thesis, we are trying to push the application "work" into the other LogGP parameters. This approach has been tried in the past in many different I/O contexts. Figure 6.1 shows the basic method behind this approach. In the context of networking, most work has used off-board protocol processors. A few designs have added a combination DMA/checksum engine [33]. More aggressive designs implemented demultiplexing and segmentation/reassembly [12].

The dangers of adding functional units to assist the main processor, an thus reduce overhead, are widely known [26, 50]. The strongest objections tend to be that assist device is "slower" in some manner. However, "slower" is often ill-defined. More precise definitions would include increased latency, reduced throughput, or even *increased* overhead because of added synchronization costs. In the next sections, we will see that although SPINE is successful at reducing overhead, it does not improve the latency or bandwidth. The importance of reducing overhead without altering the other parameters depends on the application context. For a busy server, overhead reduction might be important, but in other contexts absolute latency or bandwidth may be more critical.

In the early 1980's many commercial network adapter designs incorporated the entire protocol stack into the adapter. There were two reasons for such an approach, which, due to its complexity, required an I/O processor. First, many host operating systems did not support the range of protocols that existed at the time (e.g., TCP/IP, telnet, and rlogin) [86]. Writing these protocols once for a re-programmable network adapter was an effective method of quickly incorporating protocols into a variety of operating systems. Second, the host processors of the time were not powerful enough to run both multi-tasking jobs and network protocol stacks efficiently.

By the late 1980's however, the tide had turned, with only support for very common protocol operations included in the adapter. The migration of common protocols into commodity operating systems and the exponential growth of processor speed eliminated the original motivations for re-programmable network adapters at the time. There has been a great deal of work, however, in offloading pieces of network protocols. For example, there has been work to offload Internet checksum calculations [33], link layer processing, and packet filtering.

In terms of this thesis, there are clearly tradeoffs between reducing $o$ and increasing $L$ and $g$. For example, both the Meiko CS-2 and Paragon machines used I/O processors. Adding I/O processors added $L$ to the system, and in the Meiko they also added a very high $g$ as well [64]. Given the results of this thesis however, reducing $o$ at the expense of $L$ is the correct tradeoff. However, inflation of $g$ is not as clear a benefit as this reduces the effectiveness of latency tolerating techniques.

Although the LogGP model is quite useful, its parameters are too abstract to capture some of the performance enhancements of a system which reduces overhead. We need a method to more concretely characterize the effect of adding or removing functional units. Although we can cast such performance improvements in terms of the LogGP model, as in the above example, a better class of models are the pipeline models introduced in the next section. The problem with LogGP is that it lumps too much of the off-CPU processing into just two parameters: $g$ and $L$. In addition, these parameters include a myriad of system components. Pipeline models allow us to isolate the effect of each function unit in isolation, yet allow a re-construction of the entire communications path.

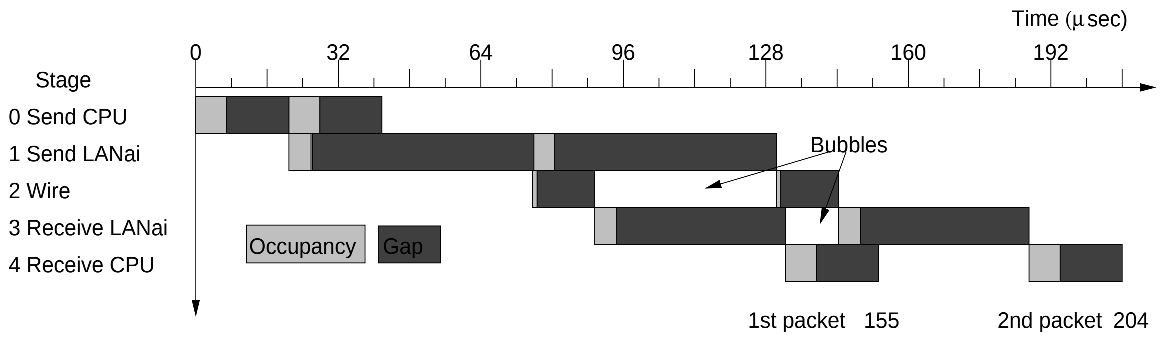


Figure 6.2: **Generic GAM Pipeline**
*This figure plots the movement of 2 packets each of size 2 KB through the abstract GAM pipeline. Time is represented on the x-axis and the stage number on the y-axis. The fixed occupancy is shown in light grey and the variable per-byte cost (Gap) time in dark grey for each stage. Bubbles (idle time) can result when moving from different speed stages.*

## 6.1 Pipeline Framework

A less common viewpoint than either queuing theoretic models or parallel program models are pipeline models [32, 106]. In this family of models, the network is modeled as a series of store-and-forward stages. The time to move data through each stage is modeled as a fixed occupancy, $O$, plus a variable per-byte cost, also called Gap, $G$. Different versions of the models arise about the restrictions placed on the stages. For example, the stages may allow for only fixed-size packets, as opposed to the more general variable size packets.

Although superficially similar to a network queuing model, the analysis techniques of pipeline models are quite different. The differences arise because the questions asked about pipelines have to do with how to discretize the packets to obtain minimum delay or maximum bandwidth through the pipeline, not steady-state behavior assuming a random process model. Some of the analysis techniques are the same as the min-max techniques used in the operations research community.

Figure 6.2 shows an abstract pipeline framework in which to reason about networking performance. Time is represented on the x-axis and stage number on the y-axis. Two 2 KB packets are shown making their way through the network pipeline. The occupancy portion of the time is represented as light grey, and the Gap portion in dark grey. Table 6.1 shows the actual values as measured in [106]. Note in the real GAM system, stages 2 and 3 are collapsed into a single stage to simplify the LANai firmware. However, performance is not affected because the sum of stages 1 and 2 nearly equals stage 1.

| Stage | Occupancy ($\mu$sec) | Gap ($\mu$sec/KB) |
|---|---|---|
| Send CPU | 6.7 | 7.2 |
| Send LANai | 5.3 | 24.5 |
| Wire | 0.2 | 6.4 |
| Recv. LANai | 5.2 | 18.5 |
| Recv. CPU | 9.6 | 7.2 |

Table 6.1: **GAM Pipeline Parameters**
*This table shows the abstract pipeline parameters for the GAM system. Each stage is abstracted by a fixed cost, called the occupancy, and a cost-per-byte, which corresponds to a Gap parameter per stage.*

## 6.2 Example: SPINE IP Router

In this section we explore the overhead reduction techniques used in the Safe Programmable Integrated Networking Environment (SPINE). SPINE allows fragments of application code to run on the network interface. An explicit goal of the SPINE system is to improve performance by reducing data and control transfers between the host and I/O device. In the context of this thesis, such reductions can reduce $o$. As we shall see in the next sections, this often comes at the expense of $L$ and $g$. The next sections show that by allowing application code to execute on the network interface, we can obtain substantial efficiencies in data movement and control transfers.

Loading application-specific code, as opposed to vendor-supplied firmware, onto a programmable adapter raises many questions. How and when does it execute? How does one protect against bugs? How does this code communicate with other modules located on the same adapter, peer adapters, remote devices, or host-based applications spread across a network?

We address these questions using extensible operating system technology derived from the SPIN operating system [14] and communication technology from the NOW project [5] to design SPINE. SPINE extends the fundamental ideas of SPIN, that is, type-safe code downloaded into a trusted execution environment. In the SPIN system, application code was downloaded into the operating system kernel. In the SPINE environment, code is downloaded into the network adapter. Extensibility is important, as we cannot predict the types of applications that may want to run directly on the adapter.

The next sections document an application we have constructed on the SPINE system: an Internet Protocol router. We have also constructed a video client application. However, the IP

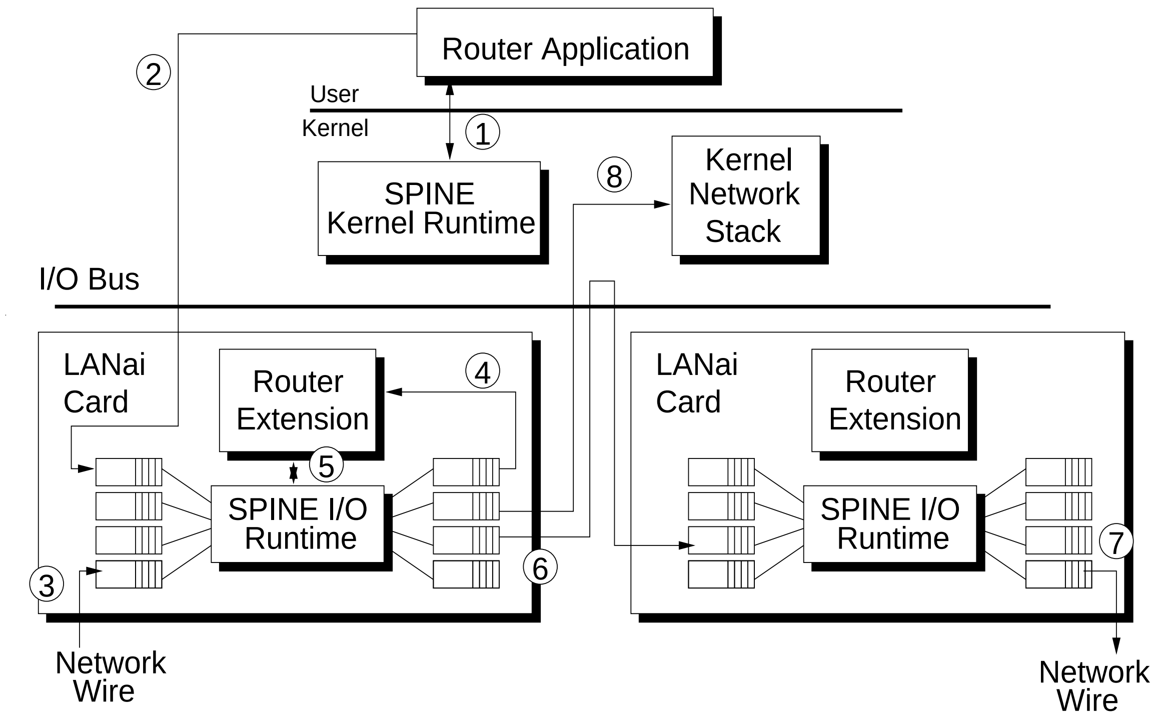


Figure 6.3: **SPINE IP Router Architecture**
*This figure shows the SPINE IP router architecture. In the common case, IP packets move directly between LANai cards, bypassing the main CPU completely.*

router is a better example of both the benefits and dangers of the overhead reduction techniques of the SPINE approach.

### 6.2.1 Architecture

The SPINE system structure is illustrated in Figure 6.3. Programming re-programmable adapters requires operating system support both on the host and on the LANai processor. A small set of core interfaces defines the SPINE *run-time*. These are implemented in C, and provide a basic execution environment (e.g. an operating system) to SPINE *extensions*. In a nutshell, SPINE extensions are application defined code to be loaded onto the network adapter. Extensions are realized as sets of Modula-3 procedures that have access to the interfaces defined by the SPINE run-time. In our case, the extensions implement IP router code.

The run-time interfaces exported to extensions include support for messaging, safe access to the underlying hardware (e.g., DMA controllers), and a subset of the Modula-3 interface. The interface also consists of message FIFOs that enable user-level applications, peer devices, and kernel modules to communicate with extensions on the network adapter using an active message style communication layer.

User-level applications inject extensions onto the adapter using SPINE's dynamic linker, and send messages directly to extensions via a memory mapped FIFO using SPINE's communication library. The kernel run-time currently works in the context of Linux and Windows NT.

### A Message-Driven Architecture

The primary purpose of a network adapter is to move messages efficiently between the system and the network media. The network adapter's basic unit of work is thus a message. To efficiently schedule messages and the associated message processing, the SPINE I/O run-time uses a message driven scheduling architecture, rather than the process or thread-oriented scheduling architecture found in conventional operating systems. The primary goal is to sustain three forms of concurrent operation: host-to-adapter message movement, message processing on the adapter, and adapter-to-network message movement.

Maximizing the number of concurrent operations is our attempt to maximize the effective bandwidth. If we blocked on each data-movement operation, each DMA engine in the LANai would see a lower effective bandwidth as it would have to "wait its turn" behind other units. In the pipelining context, we are thus attempting to maximize the effective $G$ term at the cost of increased occupancy. The increase in occupancy arises because of the overhead in managing as many concurrent operations as possible. We will see the result of this tradeoff in the next section. Briefly, we see that occupancy of the I/O processor can result in a serious performance limitation.

In SPINE the message dispatcher manages these concurrent operations. A re-programmable network adapter not only moves data; it reacts and applies transformations to it as well. On message arrival, the adapter may have to operate on the data in addition to moving it. This style of processing is captured well by the Active Message programming model, which we use to program SPINE extensions on the network adapter. Every message in the system is an active message. SPINE extensions are thus expressed as a group of active message handlers. On message arrival, the SPINE dispatcher can route messages to the host, a peer device, over the network, or invoke an active message handler of a local SPINE extension.

As Figure 6.3 shows, the I/O run-time is a glorified message multiplexor, managing the movement of messages between input queues from the host, the network, and peer LANai cards to the various output queues. The key difference in the SPINE architecture from other fast message and I/O processing schemes is that extensions provide a safe way for the user to program the interpretation and movement of messages from any of these sources.

The two goals of handler execution, in order to determine the correct action for each message, and rapid message flow implies that handlers must be short-lived. Thus, the contract between the SPINE run-time and extension handlers is that the handlers are given a small, but predictable time to execute. If a handler exceeds the threshold it is terminated in the interests of the forward progress of other messages. Long computations are possible, but must be expressed as a sequence of handlers. The premature termination of handlers opens a Pandora's box of safety issues. We do not describe these issues in this thesis, as they are tangential to the investigation of host-overhead reduction. The reader is referred to [40] for a complete treatment of the resulting operating systems issues.

To increase the amount of pipelining in the system, message processing is broken up into a series of smaller events. These events are internally scheduled as active messages, which invoke system provided handlers. Thus, only a single processing loop exists; all work is uniformly implemented as active messages. We document the performance of these internal messages, and thus the throughput and latency of the SPINE I/O run-time, in the next section.

Returning to Figure 6.3 we can trace the execution of events in the SPINE IP router. The sequence of events needed to route an IP packet from one adapter to another are:

1. The router application communicates with the SPINE kernel run-time to obtain a queue into the I/O run-time running on the LANai card.

2. The router application loads the router extension onto the LANai and then loads the IP routing tables onto the extension.

3. As a packet arrives from the wire, the hardware network DMA engine spools it to card-memory. The I/O run-time polls a set of software events; one of which polls a hardware event register. When the packet arrives, the head of the message specifies it is for the IP router extension.

4. The I/O run-time calls the router extension receive event. If no router extension is loaded, the run-time drops the packet.

5. After looking up the destination LANai of the packet, the router extension calls into the I/O run-time to send the packet to the output queue on the correct LANai card.

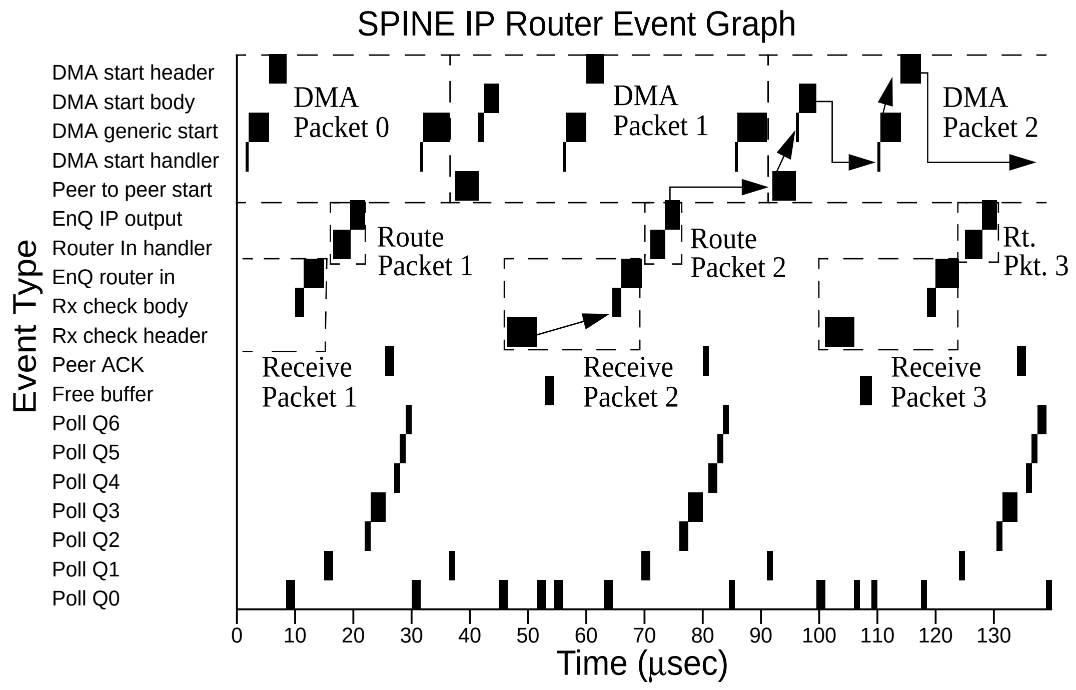


Figure 6.4: **SPINE IP Router Event Plot**
*This figure plots events as they unfold in the SPINE IP router. Time is shown on the x-axis and event types on the y-axis. A box is plotted at the point in time at each event occurrence. The width of the box corresponds to the length of the event. The dashed rectangles correspond to higher-level packet semantics: receiving a packet, routing it, and forwarding it over the PCI bus. The arrows trace the causal relationships of a single packet as it moves through the system.*

6. The I/O run-time sends the packet to the LANai card specified. In this case, the packet must move over the I/O bus into the input queue of another LANai card.

7. The second LANai card does not need to forward the packet to any extensions; the address of the output queue was known to the router extension. Thus, on finding the entire packet in the I/O bus input queue, the I/O run-time invokes the wire-output routine.

8. If the extension can not route the packet, it can be sent to the kernel network stack for further processing. Later, the host OS can route the packet to the appropriate card.

### 6.2.2 SPINE Event Processing

In this section, we show how SPINE event processing translates into overall performance. The apparatus for our experiments is different than the one used in the rest of this thesis. We use a cluster of 4 Intel Pentium Pro workstations (200MHz, 64MB memory, 512 KB L2 cache) running

Windows NT version 4.0. One node has four LANai cards and acts as the router. The Myricom LANai adapters are on a 33 MHz PCI bus, and they contain 1MB SRAM card memory. The LANai is clocked at 33 MHz, and has a wire rate of 160MB/s.

Figure 6.4 shows a real snapshot of the event processing. We used algorithms for forwarding table compression and fast IP lookup described in [79], and inserted one thousand routes into less than 20KB of memory on the adapter. The event trace was collected while forwarding two million packets at a maximum rate such that the percentage of dropped packets was less than 0.5%. The average per-packet processing time, or $g$, was 95 $\mu$s/packet.

In order to understand Figure 6.4, first recall that all SPINE processing is ultimately decomposed into internal Active Message handlers. For example, when the IP router extension calls the procedure to forward a message to a peer device, the run-time implements this functionality by issuing a series of active messages to itself. Thus, event and handler invocation becomes synonymous. This approach not only simplifies the dispatcher, but it also exposes the natural parallelism inherent in the operation.

The x-axis in Figure 6.4 represents time. The y-axis is an enumeration of event (handler) types. The dark boxes represent time spent in event processing. The position of the boxes on the x-axis shows the time a particular event took place. The position on the y-axis shows what event was processed at that time. The width of the dark box shows the length of the event. For example, during the period between 16 and 20 microseconds, the IP routing handler (Step 4 in Figure 6.3) was running.

The events are ordered by type. The lower events are polls to the 7 input queues. The next set (peer ACK and Free buffer) are LANai-to-LANai flow control over the PCI bus. Receiving over the wire occupies the next 3 events, followed by only 2 events needed to route the packet. The final 5 events manage the DMA engine over the PCI bus.

Because SPINE breaks message processing into many tiny events, the event graph at first appears to be a jumbled maze with little discernible structure. In reality however, Figure 6.4 shows a regular, periodic pattern. The key to understanding the event graph is to recognize the high level structure shown in Figure 6.3 emerging from the thicket of small events. From these patterns we can deduce both the rate at which packets are processed, as well as the latency of a packet as it passes through the system. The dashed rectangles in Figure 6.4 outline higher-level packet-processing steps. Recall that to route an IP packet requires 3 high level operations: receiving the packet (step 3 in Figure 6.3), determining the destination (routing, step 4 in Figure 6.3), and forwarding the packet to an egress adapter (steps 5-6 in Figure 6.3).

The gap, $g$, in terms of seconds per packet, is easily discernible via the period of new processing steps in the event graph. The time between receiving packets 1, 2 and 3 in Figure 6.4 is roughly 55 $\mu$s. The period for other kinds of processing, such as routing and DMA, is similar. Thus, we can conclude that we can route a new IP packet once every 55 $\mu$s.

The latency, $L$, or time it takes a single packet to move through the system, is observable by tracing the path of a single packet. The arrows in Figure 6.4 trace the processing path of packet 2 as it is received, routed, and transferred to the egress adapter. From the graph we can see that the latency of a packet through the input adapter is roughly 100 $\mu$s. Note that the bandwidth and latency are distinct due to overlap (as shown in Figure 6.4). The gap is determined by the most complex operation, which is the DMA of the packet between the two LANai cards.

A large discrepancy exists between the average measured $g$ of 95 $\mu$s/packet and the observed times in Figure 6.4 of 55 $\mu$s/packet. The key to understanding the missing 40 microseconds is that Figure 6.4 does not show a typical event ordering; rather it is a best case ordering.

We have discovered by observing many event plots that the slightly lower priority of PCI events in SPINE results in oscillations of event processing. Several incoming packets are serviced before any outgoing packets are sent over the PCI. These oscillations break an otherwise smooth packet pipeline. The system oscillates between the fast state of one-for-one receive-and-send, and the slow state of receiving a few packets and draining them over PCI. In our pipelining model, this would be modeled as a succession of occupancies followed by a several transfers, forming large pipeline bubbles. The net result is that the average forwarding time increases from 55 to 95 $\mu$s. We are currently investigating ways to improve the priority system of the SPINE dispatcher to eliminate this effect.

A closer look at Figure 6.4 shows two key limitations of the SPINE architecture. First, a general-purpose event scheduler may not always optimize performance. Second, the occupancies needed to multiplex many hardware devices on a weak embedded processor are substantial.

SPINE was constructed to be general enough so that the natural parallelism of packet processing would automatically be interleaved with user extensions. However, precise scheduling would be possible if we a priori knew what sequence of events need to be processed, and thereby achieve better overall performance. Indeed, many other projects [81, 103] have exploited this fact and developed firmware with a fixed event-processing schedule that is specialized for a specific message abstraction or application. The lack of an efficient, static processing schedule may be an inherent limitation of any general-purpose system.

The second weakness in the SPINE architecture is that the occupancy to multiplex many

| Platform | $o$ ($\mu$s) | $g$ ($\mu$s) | $L$ ($\mu$s) | MB/s($\frac{1}{G}$) |
|---|---|---|---|---|
| SPINE IP Router | 0.0 | 95 | 155 | 17.0 |
| USC/ISI IP Router | 80 | 80 | - | 16.7 |
| UltraSPARC GAM | 2.9 | 5.8 | 5.0 | 38 |

Table 6.2: **SPINE LogGP Parameters.**
*This table shows the LogGP performance of the SPINE IP router, the USC/ISI IP router, and the Berkeley GAM system. $G$ is at 2 KB packet size; larger sizes are possible for the USC/ISI router. Both the USC/ISI and SPINE routers use identical hardware, but much different software architectures. The gap of the USC/ISI router is equal to the overhead, because the CPU is the bottleneck for small packets. The gap in the SPINE router is limited by the internal scheduling algorithm of the SPINE I/O run-time. Latency results were not reported for the USC/ISI router. The table shows the SPINE safety and functionality services in the LANai significantly increase the $g$ and $L$ terms over the basic GAM parameters.*

concurrent events is substantial. In terms of our pipeline model, these occupancies show up in the fixed cost per packet, i.e., the occupancy. Each packet requires 29 events to process, resulting in a long occupancy of 100 $\mu$s per packet. Many of these events are checks for events that never occur. For example, polls to queues that are empty. However, many are more insidious forms of occupancy, such as events to manage the concurrency of the host-DMA engine.

A somewhat disappointing result is that in spite of aggressive overlap, the occupancies of the LANai processor greatly lengthen the period of packet processing. Observe how the box outlining the "DMA packet 1" in Figure 6.4 is lengthened by 10 $\mu$s due to the polls to the input queues during the period between 75-85 $\mu$s. If the LANai processor had better PCI messaging support, the period could be reduced.

The net result is that the pipeline formed by the SPINE IP router has a $g$ of within 15 $\mu$s and a slightly better $G$ than a router built from the same 200 MHz Pentium Pro processor, the same LANai cards and a modified BSD TCP/IP stack [105]. [1] Table 6.2 summarizes the LogGP parameters of the two routers. Although the SPINE architecture can obtain close to the same performance in terms of gap and latency with a weaker CPU, it is not clear that without additional architectural support, such as a faster embedded processor or additional messaging support, if the overhead reducing techniques of the SPINE architecture are worth the additional software complexity.

In the server context where high CPU utilization due to I/O results in unacceptable performance degradations, such as in Global Memory Systems [102], SPINE-like architectures make

[1] The device driver copies only the packet headers into host memory. Special code in the device driver and LCP does a direct LANai-to-LANai DMA after the regular host OS makes the forwarding decision.

sense. However, in the more general case advanced I/O architectures should improve gap and latency as well. Architectures more radical than SPINE are needed to deliver an order of magnitude performance improvement for single communication streams. A slew of novel software protocols can deliver this kind of overhead reduction [25, 72, 81, 103]. However, an unfortunate problem with these protocols is that in order to obtain their performance one loses connectivity to a vast body of applications. An open question is thus if new I/O architectures can obtain an order-of-magnitude performance improvement over traditional designs while maintaining connectivity to common protocols stacks.

# Chapter 7

# Conclusions

*Caltrans spent $1 billion to replace the old Cypress freeway. It spent millions more to widen Interstate 80. But ... the commute hasn't gotten any better. The problem is not the new Cypress freeway – it's getting to it from Berkeley and beyond.* —Catherine Bowman, SF Chronicle, Feb. 8, 1999.

*This 980 thing has been ridiculous.* —David E. Culler, SF Chronicle, Oct. 1, 1998.

This chapter concludes the thesis. We organize our conclusions around the four areas of contributions: performance analysis, observed application behavior, architecture and modeling. Each section also provides some perspective about how this thesis fits into the wider context of computer science and the sciences in general.

We conclude this chapter with a short analogy in the hope that it will help the reader remember our results. We then present some open questions and promising areas of research. We end with some final thoughts for the reader to contemplate.

## 7.1 Performance Analysis

This thesis demonstrates that performing application-centric sensitivity emulation experiments validated with analytic modeling is a powerful strategy for understanding complex computer systems. The fundamental premise of the method is that by introducing precision delays in key components we can understand their importance to overall system performance. Our perturbation method is surprising simple, almost to the point of seeming uninteresting. However, system designers use a similar style of analysis all the time in analytic modeling—so much so that the style has a name: bottleneck analysis. What makes the method in this thesis unique is that we have applied a similar methodology to real systems as opposed to analytic models or simulations.

The perturbation nature of the method has much in common with the experiments in the life-sciences. Even the smallest living organisms contain a complexity well beyond anything man-made. A fundamental question is thus how to even begin to understand such systems. Yet, many experiments in the life sciences take a similar, simple approach as in this thesis. The first step is to "damage" (in our case, slowdown) a component in a controlled manner. Next, a stimulus is applied to the system (e.g. running the application) and then the experimenter can observe differences in behavior from the baseline, "undamaged" system. Differences in behavior are thus related in some way to the modified component. An experiment in the field of neuroscience using this style of analysis can be found in [85]. It is somewhat eerie that computer systems are approaching a level of complexity that necessitates this style of analysis.

Another advantage of the method is that we can probe the system in a systematic manner without having to rely on factor analysis. This is quite different from more traditional studies comparing systems. Many studies compare two or more systems which differ from each other in many dimensions. A factor analysis must then be used to quantify the impact of each component. The method presented in this thesis takes an opposite approach. Because each factor can be controlled, we can use a single system and adjust the factors one at a time.

As computer systems become more complex, perhaps the methods used in this thesis will become more widespread. For example, often the claim is made that "the network is too slow". Using the methods in this thesis, we could slow the network down and observe if any application metrics changed. If nothing changed, we can rest assured that some other component is the bottleneck. If, on the other hand, we observe an immediate slowdown, we may conclude that improving the network will improve performance to until some other component becomes the bottleneck.

Our style of analysis is not limited to networks; we could apply our methodology to components in computer architecture, operating systems, graphics, and indeed, any computer systems area. For example, often the claim is made that the context switch time of the operating system is too slow. One could artificially inflate this time and observe the effects on application behavior in order to determine the sensitivity to context switch time. In a like manner, we could slow down the file system, virtual memory and other sub-systems in the operating system to help us isolate the impact of these systems.

The reasons for a dearth of live sensitivity approaches in the computer sciences are two-fold. First, it is a formidable engineering effort to construct a tunable apparatus. As we saw in Chapter 2, the construction requires access to components that were not designed for modification. Simulation suffers from a similar drawback in that a reasonable simulation requires substantial engineering effort. A second reason for the lack of live approaches is that the calibration can be almost as involved as building the apparatus itself. The calibration aspect cannot be taken lightly; a bug in the apparatus can easily ruin the whole experiment. The calibration aspect again mirrors experiments in other sciences, where recording calibrations can become a daily event. Fortunately, the calibrations needed in this thesis, although extensive, are not so tedious.

There is a large weakness with the method of this thesis, however. Although straightforward to observe cause and effect, without a model such data is merely adds to an increasing obscure pile of experimental results. As the quote in Chapter 5 shows, such data in an of itself is of little value unless it leads to an understanding of the how and why a system works. In this thesis, we have attempted to use as simple models as possible to understand application behavior. We discuss the effectiveness and character of our models in Section 7.4.

## 7.2 Application Behavior

The "live system" nature of our method is much more powerful than relying on analytic modeling or simulation alone, because we can make statements about application behavior. These observations, coupled with the LogGP model, allow us to draw architectural conclusions as well.

We have found that applications use a wide range of latency tolerating techniques, and that these work quite well in practice. A little reflection shows that this should not be too surprising given the tremendous attention given to latency tolerating techniques from many areas of computer science. Indeed, the 1990's has seen a broad, if somewhat disorganized, assault on the "latency" problem from the theory, language, and architecture communities. For example, models such as BSP, LogP and QSM are important theoretic tools needed to design algorithms which mask latency. Languages such as Id and Split-C were designed to allow the programmer and run time system to tolerate latency. On the architectural front, recent designs have emphasized reduced overhead and simpler interfaces to allow for greater latency tolerance (e.g. the T3D vs. T3E). The importance of these techniques across a broad range of computer science is such that they have entire book chapters devoted to the them [32].

In spite of the many latency tolerant programs, NFS does exhibit hyper-sensitivity to latency for the performance regimes of traditional LANs. However, as was noted in Chapter 5.6, IP switching hardware has crossed a fundamental threshold into the 10-20 microsecond regime. Although these latencies are still an order of magnitude higher than current SANs, they push NFS into a latency-insensitive region.

Of the applications studied, the NPB represent the least sensitive to all the LogGP parameters. This is partly due to the combination of scaling rule (fixed problem size) and machine size used. On a 32 processor machine the communication to computation ratio is quite small, even with the larger class B problem size. Given fixed problem-size scaling, only on very large configurations, e.g. 512 processors, will any of the LogGP parameters have much impact on overall performance.

A second reason the NPB are quite insensitive to communication is because these codes have been extensively studied at the algorithmic level. Over the past 10 years much attention has been focused on how to minimize communication costs in these codes. Given that the early parallel machines that the NPB were developed on (e.g. the nCUBE/2 and the iPSC/1) had very large communication costs (5,000+ cycles) it is not too surprising that much attention was given to minimizing the costs of communication. In fact, literature from the 1980's often models all communication as pure overhead, because message passing machines at the time provided little opportunity for overlap [42].

It is interesting to conjecture if the application behavior observed is fundamental to the application, or simply a historical accident. The developers of each suite certainly had an architectural model in mind when designing the applications measured in this thesis, and this is reflected in the structure of the applications. For the Split-C/AM benchmarks, the model was low-overhead parallel machines and clusters. The NPB were designed in the context of previous generation high-overhead hypercubes. NFS was developed in the LAN context. Perhaps the only certain claim we can make about the "fundamental" properties of these applications is that with the passage of time, programmers will invent new ways to tolerate latency and avoid overhead. The clever application designer is rewarded for shifting the sensitivity of the application away from $L$, avoiding $o$, and towards $g$ and $G$ wherever possible. A common pattern is that as applications age, they first lose sensitivity to latency, then to overhead, and finally end with some sensitivity to a form of bandwidth (either $g$ or $G$).

## 7.3 Architecture

The primary architectural result of this thesis is that software overheads for communication performance are still too high. Of all the LogGP parameters, the sensitivity to $o$ cannot be overstated. This is because many of the latency-tolerating techniques are still sensitive to overhead. For example, work overlapping and communication pipelining techniques still incur a cost of $o$ on every message, even though they can mask latency.

Even for Split-C/AM applications, which were developed on low-overhead machines, overhead is still a limiting factor. Sensitivities slopes of 1-2 were common for the Split-C/AM programs. For NFS, we observed a sensitivity slope of -1.5 in overhead vs. throughput. These sensitivities also do not have flat regions, implying that further reductions in overhead will have immediate benefits. We observed some of the benefits of reduced overhead for NFS in the Network Appliance box; that machine can sustain a much higher throughput than a comparable box running OSF/1 by using a specialized operating system.

The NAS Parallel Benchmarks did not exhibit much sensitivity to overhead. This result is clearly explainable by observing the applications' structure; few messages are sent for the machine size (32) and problem size (class B) used in this study. Our results might be different if we extrapolated to an order-of-magnitude change in machine size, i.e., a 512-node machine. However, such machines are the uncommon case. Much as they have in the past, "small" configurations of 32 and 64 nodes will continue to dominate the field of parallel computing.

Almost all of the latency tolerating techniques of applications shift the sensitivity from latency to some form of bandwidth, either per-message as in $g$ or per-byte, $G$. The pressures that latency tolerating techniques place on bandwidth are not unique to networking, they have been observed in the CPU regime as well [20]. Although maintaining a high per-byte bandwidth is quite tractable, obtaining high per-message bandwidths is still an architectural challenge.

Our results on application behavior lead us to the somewhat counterintuitive architectural conclusion that future architectures do not need to continue to optimize for network transit latency. Instead, designers should focus on reducing overhead. Programmers are adept at using latency-tolerating techniques, thus architectures should focus on enabling programmers to better tolerate latency. From an architectural perspective, building machines to tolerate latency is easier than reducing the entire end-to-end path. In practice, it means designers should concentrate on improving access to the network interface while maintaining a high message rate; many latency tolerating techniques are still sensitive to overhead and gap.

A host of novel techniques exist to reduce both overhead and gap in the network interface hardware/software combination. However, the problem with non-standard techniques is that they ignore a very large existing infrastructure which is unlikely to change for the foreseeable future. Powerful non-technical forces will continue to cause large software overheads. Intuitively, the network interface is where three vendors' software and hardware must work together: the operating system vendor, the switch/hub vendor, and the network interface hardware vendor. Immutable standards for connecting all three are thus inevitable. For example, porting NFS to any of the alternative message passing layers is not impossible, but is certainly a formidable engineering task. The challenge to future network interface designers will be to reduce overhead and gap while maintaining connectivity between applications in the existing infrastructure.

One might be tempted to simply add CPU's and network interfaces in a large SMP box to decrease the effective gap and Gap, or to amortize the overhead among many processors. However, such an approach has several limitations. First, the parallelization of a single stream is quite limited using current operating systems [91]. Thus, In order to obtain a reduced gap, the application has to parallelize the communication into multiple streams itself. Second, the size of the machine needed to sustain a very high effective $g$ and $G$ is substantial. For example, in order to add just 8 gigabit network interfaces into a server and use them simultaneously requires 8 separate I/O busses. While machines of this size do exist, the very high premium attached to this class of machines is well-known.

## 7.4 Modeling

We have found that simple models can give "reasonable" performance predictions. The simple frequency-cost pair overhead models were often close to the measured performance. At worst they were 50% inaccurate. The results for gap were farther off, and for latency the results are even more inaccurate. From an architectural standpoint, these results shows that a simple frequency-cost pair analysis is an adequate "ballpark" measure for a system designer. However, more detailed application and system models are needed (e.g. [39, 43]) to make truly accurate predictions across a range applications and machine configurations.

The simple models proved useful in evaluating assumptions about application behavior. For example, the simple gap models showed that communications are bursty in nature for both the Split-C/AM programs and NPB. These models also showed that serial dependencies can cause hyper-sensitives to overhead. The radix sort is a prime example of this effect.

The queuing models used for the NFS system are much more accurate than simple frequency cost-pair models used for the other applications. This accuracy, however, is somewhat circular. The SPECsfs benchmark is built using some assumptions of queuing theory, namely the traffic is generated as a Poisson process. Given that observed traffic is quite bursty, we would expect actual NFS traffic to be more sensitive to overhead and Gap than our results showed. However, given that the observed sensitivity to $G$ is very low, even under the worst-case assumption that all messages are sent in bursts, the small nature of observed NFS requests means that even current LANs will not bandwidth limit NFS.

The one place where the queuing model proved quite useful was in interpreting the results of vendors' SFS curves. Section 5.6 gave a small example of how we could compare two servers given SFS curves and the SPECsfs disclosures. We saw that we could, for example, derive the total software overhead from observing the base, slope and saturation points. An interesting exercise would be to see how well the model did on a variety of published curves. However, such a comparison is beyond the scope of this thesis.

## 7.5 Final Thoughts

Modern computer systems have reached mind boggling complexity. The design of a modern business server includes sub-systems that are impressive engineering achievements in their own right: the processor, the memory and I/O system, the operating system, the database and the business application logic.

To make the example more concrete, imagine the number of designers involved in a 4-way UltraSPARC III server, running Solaris 7, Oracle 8 and SAP R/3 on 18 GB IBM disk drives, stitched together with the UPA memory bus, multiple PCI and SCSI busses, and connected to the outside world via Alteon gigabit Ethernets (each with 2 processors). The number of people involved in the entire design certainly ranges into the tens of thousands. No one person can hope to understand it all. Yet, performance analysis of such systems is not an impossible task.

The staggering complexity of such systems will require computer performance analysts to increasingly use "black-box" methods. In this thesis we investigated one such method in the context of computer networks. Similar analysis techniques will eventually become an accepted methodology in computer science, much as they have in the other sciences.

With regard to our findings, we leave the reader with a short analogy in the hope it will serve as an aid to recalling our experimental results. The quotations at the beginning of this chapter parallels an everyday experience many drivers have in the San Francisco Bay Area have to that of modern computer networks. Commuters often wonder why after opening the billion dollar Cypress freeway, congestion seems just as bad as when using the previous detour, Interstate 980. On a smaller scale, computer users wonder why, after installing their new gigabit networks, applications don't seem any faster. In both cases, it's not the freeway or network that is the limiting factor per say. Rather, it's the access to the network or freeway: software overhead in computer networks, on-ramps in the freeway case, that are the real bottlenecks.

The SPINE work showed some of the benefits and costs of using more specialized software to reduce overhead. Although quite successful at reducing overhead, the resulting pipeline was not faster in terms of gap or latency than a fast CPU running a more standard, but still modified, TCP/IP stack. Chapter 6 showed that it an open question is if the overhead reduction obtainable with novel SAN protocols can be achieved with the much more common Internet protocols.

In the final analysis, we can conclude from the results in this thesis that computer systems are complex enough to warrant our controlled perturbation, emulation-based methodology. We observed that programmers used a variety of latency tolerating techniques and that these work quite well in practice. However, many of these techniques are still sensitive to software overhead. We found that without either more aggressive hardware support or the acceptance of radical new protocols, software overheads will continue to limit communication performance across a wide variety of application domains.

# Bibliography

[1] AHN, J. S., DANZIG, P. B., LIU, Z., AND YAN, L. Evaluation of TCP Vegas: emulation and experiment. In *Proceedings of the ACM SIGCOMM '95 Conference on Communications Architectures and Protocols* (Cambridge, MA, Aug. 1995).

[2] ALEXANDROV, A., IONESCU, M., SCHAUSER, K. E., AND SCHEIMAN, C. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures* (May 1995).

[3] ANDERSON, E. A., AND NEEFE, J. M. An Exploration of Network RAM. Tech. Rep. CSD-98-1000, University of California at Berkeley, July 1998.

[4] ANDERSON, J.-A. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T., SITES, R. L., VANDERVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (1997), pp. 1–14.

[5] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A Case for NOW (Networks of Workstations). *IEEE Micro* (Feb. 1995).

[6] ARGONNE NATIONAL LABORATORY. *MPICH-A Portable Implmentation of MPI*, 1997. http://www.mcs.anl.gov/mpi/mpich.

[7] ARPACI, R. H., CULLER, D. E., KRISHNAMURTHY, A., STEINBERG, S., AND YELICK, K. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture* (1995).

[8] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. High-performance sorting on networks of workstations. In *In Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (Tucson, AZ, May 1997), pp. 243–254.

[9] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAM, V., AND WEERATUNGA, S. K. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[10] BAILEY, D. H., BARSZCZ, E., DAGUM, L., AND SIMON, H. D. NAS Parallel Benchmark Results. Tech. Rep. RNR-93-016, NASA Ames Research Center, 1993.

[11] BAILEY, D. H., HARRIS, T., DER WIGNGAART, R. V., SAPHIR, W., WOO, A., AND YARROW, M. The NAS Parallel Benchmarks 2.0. Tech. Rep. NAS-95-010, NASA Ames Research Center, 1995.

[12] BAILEY, M. L., PAGELS, M. A., AND PETERSON, L. L. The *x*-chip: An Experiment in Hardware Demultiplexing. In *Proceedings of the IEEE Workshop on High Performance Communications Subsystems* (Feb. 1991).

[13] BARSZCZ, E., FATOOHI, R., VENKATKRISHNAN, V., AND WEERATUNGA, S. Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors. Tech. Rep. RNR-93-007, NASA Ames Research Center, Apr. 1993.

[14] BERSHAD, B. N., CHAMBERS, C., EGGERS, S., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. SPIN—An Extensible Microkernel for Application-Specific Operating System Services. Tech. rep., University of Washingtion, 1994.

[15] BLACK, R., LESLIE, I., AND MCAULEY, D. Experience of Building an ATM switch for the Local Area. In *Proceedings of the ACM SIGCOMM '94 Conference on Communications Architectures and Protocols* (London, UK, Sept. 1994), pp. 158–167.

[16] BLUMRICH, M. A., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E., AND SANDBERG, J. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture* (Apr. 1994).

[17] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. Myrinet—A Gigabit-per-Second Local-Area Network. *IEEE Micro 15*, 1 (Feb. 1995), 29–38.

[18] BREWER, E. A. High-Level Optimization via Automated Statistical Modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (June 1995).

[19] BREWER, E. A., AND KUSZMAUL, B. C. How to Get Good Performance from the CM-5 Data Network. In *Proceedings Eighth International Parallel Processing Symposium (SPAA)* (Cancun, MX, Apr. 1994).

[20] BURGER, D., GOODMAN, J. R., AND KAGI, A. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture* (Philadelphia, PA, May 1996), pp. 78–89.

[21] CHANG, K., MORRIS, R., AND KUNG, H. T. NFS Dynamics Over Flow-Controlled Wide Area Networks. In *Proceedings of the 1997 INFOCOMM* (Kobe, Japan, Apri 1997), pp. 619–625.

[22] CHESAPEAKE COMPUTER CONSULTANTS, INC. *Test TCP (TTCP)*, 1997. http://www.ccci.-com/tools/ttcp/.

[23] Chiou, D., Ang, B., Arvind, Beckerle, M., Boughton, G., Greiner, R., Hicks, J., and Hoe, J. StarT-NG: Delivering Seamless Parallel Computing. In *EURO-PAR'95 Conference* (Aug. 1995).

[24] Chiu, D. M., and Jain, R. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems 17* (1989), 1–14.

[25] Chun, B. N., Mainwaring, A. M., and Culler, D. E. Virtual Network Transport Protocols for Myrinet. *IEEE Micro 18*, 1 (1998), 53–63.

[26] Clark, D. D., Jacobson, V., Romkey, J., and Salwen, H. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine 6* (June 1989), 23–29.

[27] Cmelik, B., and Keppel, D. Shade: a Fast Instruction-set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference* (May 1994).

[28] Culler, D. E., Dusseau, A. C., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93* (1993), pp. 262–273.

[29] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), pp. 262–273.

[30] Culler, D. E., Keeton, K. K., Liu, L. T., Mainwaring, A. M., Martin, R. P., Rodrigues, S., Wright, K., and Yoshikawa, C. O. The Generic Active Message Interface Specification. NOW Research Project White Paper, http://now.cs.berkeley.edu/Papers2, 1995.

[31] Culler, D. E., Liu, L. T., Martin, R. P., and Yoshikawa, C. O. Assessing Fast Network Interfaces. In *IEEE Micro* (Feb. 1996), vol. 16, pp. 35–43.

[32] Culler, D. E., Singh, J. P., and Gupta, A. *Parallel Computer Architecture: A Harware/Software Approach*. Morgan Kaufmann, 1999.

[33] Dalton, C., Watson, G., Banks, D., and Calamvokis, C. Afterburner (network-independent card for protocols). *IEEE Network 3*, 4 (July 1993), 36–43.

[34] Dill, D., Drexler, A., Hu, A., and Yang, C. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design: VLSI in Computers and Processors* (1992).

[35] Dongarra, J. J., and Dunigan, T. Message-Passing Performance of Various Computers. Tech. Rep. UT-CS-95-299, University of Tennessee, Knoxville, July 1995.

[36] Dongarra, J. J., and Dunigan, T. *MPI Benchmark*, May 1995. http://www.netlib.org.-benchmark/comm.tgz.

[37] DUBE, R., RAIS, C. D., AND TRIPATHI, S. Improving NFS Performance Over Wireless Links. *IEEE Transactions on Computers 46*, 3 (Mar. 1997), 290–298.

[38] DUCHAMP, D. Optimistic Lookup of Whole NFS Paths in a Single Operation. In *Proceedings of the 1994 USENIX Summer Conference* (Boston, MA, June 1994), pp. 161–169.

[39] DUSSEAU, A. C., CULLER, D. E., SCHAUSER, K. E., AND MARTIN, R. P. Fast Parallel Sorting Under LogP: Experience with the CM-5. In *IEEE Transactions on Parallel and Distributed Systems* (1996), vol. 7, pp. 791–805.

[40] FIUCZYNSKI, M. E., MARTIN, R. P., BERSHAD, B. N., AND CULLER, D. E. SPINE: An Operating System for Intelligent Network Adapters. Tech. Rep. UW-CSE-98-08-01, University of Washington, Aug. 1998.

[41] FOX, A., GRIBBLE, S., CHAWATHE, Y., AND BREWER, E. Scalable Cluster-Based Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Oct. 1997).

[42] FOX, G. C., JOHNSON, M. A., LYNENGA, G. A., OTTO, S. W., SALMON, J. K., AND WALKER, D. W. *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*. Prentice Hall, 1988.

[43] FRANK, M. I., AGARWAL, A., AND VERNON, M. LoPC: Modeling Contention in Parallel Algorithms. In *Proceedings of SIXTH ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (June 1997).

[44] FROESE, K. W., AND BUNT, R. B. The Effect of Client Caching on File Server Workloads. In *Proceedings of the 29th Hawaii International Conference on System Sciences* (Wailea, HI, Jan. 1996), pp. 150–159.

[45] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., VAHDAT, A. M., AND ANDERSON, T. E. GLUnix: a Global Layer Unix for a Network of Workstations. *Software Practice and Experience 28*, 9 (July 1998), 929–61.

[46] GIBBONS, P., MATIAS, Y., AND RAMACHANDRAN, V. Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? In *Symposium on Parallel Algorithms and Architectures* (July 1997), pp. 72–83.

[47] GILLETT, R. B. Memory Channel Network for PCI. In *IEEE Micro* (Feb. 1996), vol. 16, pp. 12–18.

[48] GUSELLA, R. A Measurement Study of Diskless Workstation Traffic on an Ethernet. *IEEE Transactions on Communications 38*, 9 (Sept. 1990), 1557–1568.

[49] HALL, J., SABATINO, R., CROSBY, S., LESLIE, I., AND BLACK, R. Counting the Cycles: a Comparative Study of NFS Performance Over High Speed Networks. In *Proceedings of the 22nd Annual Conference on Local Computer Networks (LCN'97)* (Minneapolis, MN, Nov. 1997), pp. 8–19.

[50] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[51] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter 1994 USENIX Conference* (San Francisco, CA, Jan. 1994), pp. 235–246.

[52] HOLT, C., HEINRICH, M., SINGH, J. P., ROTHBERG, E., AND HENNESSY., J. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Tech. Rep. CSL-TR-95-660, Stanford University, Jan. 1995.

[53] HORST, R. TNet: A Reliable System Area Nework. *IEEE Micro 15*, 1 (Feb. 1995), 37–45.

[54] HUANG, P., ESTRIN, D., AND HEIDEMANN, J. Enabling Large-scale Simulations: Selective Abstraction Approach to the Study of Multicast Protocols. In *In Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98)* (Montreal, July 1998).

[55] IANNELLO, G., LAURIA, M., AND MERCOLINO, S. LogP Performance Characterization of Fast Messages atop Myrinet. In *Sixth Euromicro Workshop on Parallel and Distributed Processing (PDP'98)* (Madrid, Spain, Jan. 1998), pp. 395–401.

[56] JACOBSON, V. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM '88 Conference on Communications Architectures and Protocols* (Stanford, CA, Aug. 1988), pp. 314–329.

[57] JAIN, R. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[58] JONES, R. Netperf 2.1 Homepage. http://www.cup.hp.com/netperf/NetperfPage.html, Feb. 1995.

[59] JUSZCZAK, C. Improving the Write Performance of an NFS Server. In *Proceedings of the 1994 USENIX Winter Conference* (Jan. 1994).

[60] KAY, J., AND PASQUALE, J. The Importance of Non-Data-Touching Overheads in TCP/IP. In *Proceedings of the 1993 SIGCOMM* (San Francisco, CA, September 1993), pp. 259–268.

[61] KEETON, K., PATTERSON, D. A., AND ANDERSON, T. E. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III* (Stanford University, Stanford, CA, August 1995).

[62] KHALIL, K. M., LUC, K. Q., AND WILSON, D. V. LAN Traffic Analysis and Workload Characterization. In *Proceedings of the 15th Conference on Local Computer Networks* (Minneapolis, MN, Sept. 1990), pp. 112–122.

[63] KLEINROCK, L. *Queueing Systems*. John Wiley & Sons, New York, 1976.

[64] KRISHNAMURTHY, A., SCHAUSER, K. E., SCHEIMAN, C. J., WANG, R. Y., CULLER, D. E., AND YELICK, K. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1997), pp. 37–48.

[65] LAZOWSKA, E. D., ZAHORIAN, J., GRAHAM, G. S., AND SEVCIK, K. C. *Quantitative System Performance : Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, N.J, 1984.

[66] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the Self-Similar Nature of Ethernet Traffic. *IEEE Transactions on Networking 2*, 1 (Feb. 1994), 1–15.

[67] LELAND, W. E., AND WILSON, D. V. High Time-Resolution Measurement and Analysis of LAN traffic: Implications for LAN Interconnection. In *Proceedings of the 1991 INFOCOMM* (Bal Harbour, FL, April 1991), pp. 1360–1366.

[68] LIU, J. C. S., SO, O. K. Y., AND TAM, T. S. NFS/M: an open platform mobile file system. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (Amsterdam, Netherlands, May 1998), pp. 488–495.

[69] LUMETTA, S. S., KRISHNAMURTHY, A., AND CULLER, D. E. Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines. In *Proceedings of Supercomputing '95* (1995).

[70] MACKLEM, R. Lessons Learned Tuning the 4.3 BSD Reno Implementation of the NFS Protocol. In *Proceedings of the 1991 USENIX Winter Conference* (Jan. 1991), pp. 53–64.

[71] MACKLEM, R. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the 1994 USENIX Winter Conference* (Jan. 1994), pp. 261–278.

[72] MARTIN, R. P. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of the 2nd Hot Interconnects Conference* (July 1994).

[73] MARTIN, R. P., VAHDAT, A. M., CULLER, D. E., AND ANDERSON, T. P. The Effects of Latency, Overhead and Bandwidth in a Cluster of Workstations. In *Proceedings of the 24th International Symposium on Computer Architecture* (Denver, CO, June 1997).

[74] MCVOY, L., AND STAELIN, C. lmbench: Portable Tools for Performance Analysis . In *Proceedings of the 1996 USENIX Conference* (Jan. 1996).

[75] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message Passing Interface Standard Version 1.1*, July 1995. http://www.mcs.anl.govcom/mpi.

[76] MOGUL, J. C. Network Locality at the Scale of Processes. *ACM Transactions on Computer Systems 10*, 2 (May 1992), 81–109.

[77] MOGUL, J. C. Recovery in Spritely NFS. *Computing Systems 7*, 2 (1994), 201–62.

[78] MORITZ, C. A., AND FRANK, M. I. LoGPC: Modeling Network Contention in Message-Passing Programs. In *Proceedings of the 1998 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems* (Madison, WI, June 1998).

[79] NILSSON, S., AND KARLSSON, G. Fast Address Lookup for Internet Routers. In *Fourth International Conference on Broadband Communications* (Stuttgart, Germany, Apr. 1998), pp. 11–22.

[80] OUSTERHOUT, J. K. Personal communication, Jan. 1997.

[81] PAKIN, S., LAURIA, M., AND CHIEN, A. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95* (San Diego, California, 1995).

[82] PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D., AND HITZ, D. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Conference* (Boston, MA, June 1994), pp. 137–152.

[83] PAYROUZE, N., AND MULLER, G. FT-NFS: an Efficient Fault-Tolerant NFS Server Designed for Off-the-Shelf Workstations. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing* (Sendai, Japan, June 1996), pp. 64–73.

[84] PFISTER, G. F., AND NORTON, V. A. Hot Spot Contention and Combining Multistage Interconnection Networks. *IEEE Transactions on Computers C-34*, 10 (1985), 943–8.

[85] PHAN, M. L., SCHENDEL, K. L., RECANZONE, G. H., AND ROBERTSON, L. C. Acoustic Spatial Deficits in a Patient with Bilateral Parietal Damage. In *27th Annual Meeting of the Society for Neuroscience* (New Orleans, LA, Oct. 1997), p. 1312.

[86] POWERS, G. A front-end TELNET/rlogin Server Implementation. In *UniForum 1986 Conference Proceedings* (Anaheim, CA, Feb. 1986), pp. 27–40.

[87] REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, May 1993), pp. 48–60.

[88] REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture* (Apr. 1994), pp. 325–336.

[89] ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. Complete Computer Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology* (Fall 1995).

[90] SAAVEDRA-BARRERA, R. H. CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking. Tech. Rep. CSD-92-684, University of California at Berkeley, Feb. 1992.

[91] SCHMIDT, D. C., AND SUDA, T. Measuring the Performance of Parallel Message-based Process Architectures. In *Proceedings of the 1995 INFOCOMM* (1995).

[92] SCHNARR, E., AND LARUS, J. R. EEL: Machine-independent Executable Editing. In *In Proceedings of the 1995 ACM Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, CA, June 1995).

[93] SHEIN, B., CALLAHAN, M., AND WOODBURY, P. NFSSTONE - A Network File Server Performance Benchmark. In *Proceedings of the 1989 USENIX Summer Conference* (Baltimore, MD, June 1989), pp. 269–274.

[94] STANDARD PERFORMANCE EVALUATION CORP. *SPEC SFS97 Benchmarks*, 1997. http://-www.specbench.org/osg/sfs97.

[95] STANDARD PERFORMANCE EVALUATION CORP. *SPECsfs97 Press Release Results*, 1997. http://www.specbench.org/osg/sfs97/results.

[96] STERN, H. L., AND WONG, B. L. NFS Performance and Network Loading. In *Proceedings of the Sixth Systems Administration Conference (LISA VI)* (Oct. 1992).

[97] STERN, U., AND DILL, D. L. Parallelizing the Murphi Verifier. In *9th International Conference on Computer Aided Verification* (May 1997), pp. 256–267.

[98] STRATEGIC NETWORKS CONSULTING. *Fore Systems Intelligent Gigabit Routing Switch Custom Test*, Oct. 1998. http://www.snci.com/reports/ESX-4800.pdf.

[99] STRATEGIC NETWORKS CONSULTING. *Packet Engines PowerRail 5200 Enterprise Routing Switch Custom Test*, Apr. 1998. http://www.snci.com/reports/packetengines.pdf.

[100] VAHALIA, U., GRAY, C. G., AND TING, D. Metadata logging in an NFS server. In *Proceedings of the 1995 USENIX Winter Conference* (Jan. 1995), pp. 265–276.

[101] VALLIANT, L. G. A Bridging Model for Parallel Computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[102] VOELKER, G. M., JAMROZIK, H. A., VERNON, M. K., LEVY, H. M., AND LAZOWSKA, E. Managing server load in global memory systems. In *Proceedings of the 1997 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems* (June 1997).

[103] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP* (Copper Mountain, CO, December 1995), pp. 40–53.

[104] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture* (May 1992).

[105] WALTON, S., HUTTON, A., AND TOUCH, J. High-speed Data Paths in Host-based Routers. *Computer 31*, 11 (1998), 46–52.

[106] WANG, R., KRISHNAMURTHY, A., MARTIN, R. P., ANDERSON, T., AND CULLER, D. E. Modeling and Optimizing Communication Pipelines. In *Proceedings of the 1998 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems* (Madison, WI, June 1998).

[107] WITTLE, M., AND KEITH, B. E. LADDIS: the Next Generation in NFS File Server Benchmarking. In *Summer 1993 USENIX Conference* (Cincinnati, OH, June 1993), pp. 111–128.

[108] WONG, B. *Configuration and Capacity Planning for Solaris Servers*. Prentice-Hall, 1997.

[109] WONG, F. C. Message Passing Interface on NOW Performance. http://www.cs.berkeley.-edu/Fastcomm/MPI/performance, 1997.

[110] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995), pp. 24–36.

[111] YARROW, M., AND DER WIJNGAART, R. V. Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes. Tech. Rep. NAS-97-032, NASA Ames Research Center, Nov. 1997.

# Appendix A

# SPECsfs97 Disclosures

This appendix has the SPECsfs97 disclosures. Two sets of disclosures are provided, one for the SCSI system and one for the RAID system. The purposed of the disclosure is to allow others to verify results. The format for the tables was taken from the SPECsfs97 webpages. Not all data points for all experiments are listed; only the baseline numbers are provided (the measured lines in Table 5.3).

| Throughput (Ops/sec) | Response Time msec |
|---|---|
| 198 | 7.3 |
| 397 | 10.1 |
| 602 | 13.8 |
| 800 | 15.6 |
| 950 | 17.6 |
| 1006 | 18.9 |
| 1053 | 19.9 |

Table A.1: **SPECsfs97 Disclosure: SCSI Performance**

| **Configuration** | | | |
|---|---|---|---|
| **Server Configuration and Availability** | | **Network Subsystem** | |
| Vendor | Sun Microsystems Inc. | Network | Myrinet |
| Hardware Available | Mar 1995 | Controller Desc. | Myricom 128K LANai |
| Software Available | Oct 1997 | Number Networks | 1 |
| Date Tested | May 1998 | Number Network Controllers | 1 |
| SFS License Number | A-8 | Protocol Type | UDP |
| Licensee Locations | Berkeley, CA | Switch Type | Myricom 8x160 MBs |
| **CPU, Memory and Power** | | Bridge Type | N/A |
| Model Name | Sun Ultra-1 | Hub Type | N/A |
| Processor | 167 MHz UltraSPARC-1 | Other Network Hardware | N/A |
| Primary Cache | 16KB I+16KB D on chip | **Disk Subsystem and Filesystems** | |
| Secondary Cache | 512K(I+D) off chip | Number Disk Controllers | 3 |
| Other Cache | N/A | Number of Disks | 25 |
| UPS | N/A | Number of Filesystems | 25 |
| Other Hardware | N/A | File System Creation Ops | default |
| Memory Size | 128 MB | File System Config | default |
| NVRAM Size | N/A | Disk Controller | On-board narrow SCSI |
| NVRAM Type | N/A | # of Controller Type | 1 |
| NVRAM Description | N/A | Number of Disks | 1 |
| **Server Software** | | Disk Type | 2GB 5200RPM SCSI |
| OS Name and Version | Solaris 2.5.1 | File Systems on Disks | OS, swap |
| Other Software | Myrinet-GAM device drv. | Special Config Notes | N/A |
| File System | UFS | Disk Controller | Sun FAS wide SCSI |
| NFS version | 2 | # of Controller Type | 2 |
| **Server Tuning** | | Number of Disks | 24 |
| Buffer Cache Size | Dynamic | Disk Type | 9GB 7200RPM IBM SCSI |
| # NFS Processes | 128 | File Systems on Disks | F1-F24 |
| Fileset Size | 8.7 GB | Special Config Notes | N/A |

Table A.2: **SPECsfs97 Disclosure: SCSI Server and Network**

| Load Generator (LG) Configuration | |
|---:|---:|
| Number of Load Generators | 3 |
| Number of Processes per LG | 7 |
| Biod Max Read Setting | 5 |
| Biod Max Write Setting | 5 |
| LG Type | LG1 |
| LG Model | Ultra1-170 |
| Number and Type Processors | 1 167MHz UltraSPARC |
| Memory Size | 128MB |
| Operating System | Solaris 2.5.1 |
| Compiler | gcc |
| Compiler Options | -O2 |
| Network Type | Myricom 128K LANai |

Table A.3: **SPECsfs97 Disclosure: SCSI Load Generators**

| **Testbed Configuration** | | | | |
|---|---|---|---|---|
| 1 | LG1 | N1 | F1..F8 | N/A |
| 2 | LG1 | N1 | F9..F16 | N/A |
| 3 | LG1 | N1 | F17..F24 | N/A |

Table A.4: **SPECsfs97 Disclosure: SCSI Testbed Configuration**

| Throughput (Ops/sec) | Response Time msec |
|---:|---:|
| 298 | 3.4 |
| 397 | 4.0 |
| 494 | 4.5 |
| 595 | 4.9 |
| 696 | 6.1 |
| 797 | 7.2 |
| 901 | 8.8 |
| 1001 | 11.4 |
| 1205 | 15.0 |
| 1304 | 18.7 |
| 1405 | 21.0 |
| 1503 | 30.8 |

Table A.5: **SPECsfs97 Disclosure: RAID Performance**

| **Configuration** | | | |
|---|---|---|---|
| **Server Configuration and Availability** | | **Network Subsystem** | |
| Vendor | Sun Microsystems Inc. | Network | Myrinet |
| Hardware Available | Mar 1995 | Controller Desc. | Myricom 128K LANai |
| Software Available | Oct 1997 | Number Networks | 1 |
| Date Tested | May 1998 | Number Network Controllers | 1 |
| SFS License Number | A-8 | Protocol Type | UDP |
| Licensee Locations | Berkeley, CA | Switch Type | Myricom 8x160 MBs |
| **CPU, Memory and Power** | | Bridge Type | N/A |
| Model Name | Sun Ultra-1 | Hub Type | N/A |
| Processor | 167 MHz UltraSPARC-1 | Other Network Hardware | N/A |
| Primary Cache | 16KB I+16KB D on chip | **Disk Subsystem and Filesystems** | |
| Secondary Cache | 512K(I+D) off chip | Number Disk Controllers | 2 |
| Other Cache | N/A | Number of Disks | 29 |
| UPS | N/A | Number of Filesystems | 16 |
| Other Hardware | N/A | File System Creation Ops | default |
| Memory Size | 128 MB | File System Config | default |
| NVRAM Size | N/A | Disk Controller | On-board narrow SCSI |
| NVRAM Type | N/A | # of Controller Type | 1 |
| NVRAM Description | N/A | Number of Disks | 1 |
| **Server Software** | | Disk Type | 2GB 5200RPM SCSI |
| OS Name and Version | Solaris 2.5.1 | File Systems on Disks | OS, swap |
| Other Software | Myrinet-GAM device drv. | Special Config Notes | N/A |
| File System | UFS | Disk Controller | Sun A3000 RAID |
| NFS version | 2 | # of Controller Type | 1 |
| **Server Tuning** | | Number of Disks | 28 |
| Buffer Cache Size | Dynamic | Disk Type | 9GB 7200RPM Seagate SCSI |
| # NFS Processes | 128 | File Systems on Disks | F1-F15 |
| Fileset Size | 8.7 GB | Special Config Notes | N/A |

Table A.6: **SPECsfs97 Disclosure: RAID Server and Network**

| Load Generator (LG) Configuration | |
|---|---|
| Number of Load Generators | 3 |
| Number of Processes per LG | 15 |
| Biod Max Read Setting | 3 |
| Biod Max Write Setting | 2 |
| LG Type | LG1 |
| LG Model | Ultra1-170 |
| Number and Type Processors | 1 167MHz UltraSPARC |
| Memory Size | 128MB |
| Operating System | Solaris 2.5.1 |
| Compiler | gcc |
| Compiler Options | -O2 |
| Network Type | Myricom 128K LANai |

Table A.7: **SPECsfs97 Disclosure: RAID Load Generators**

| Testbed Configuration | | | | |
|---|---|---|---|---|
| LG # | LG Type | Network | Target File Systems | Notes |
| 1 | LG1 | N1 | F1..F15 | N/A |
| 2 | LG1 | N1 | F1..F15 | N/A |
| 3 | LG1 | N1 | F1..F15 | N/A |

Table A.8: **SPECsfs97 Disclosure: RAID Testbed Configuration**

# Appendix B

# Performance Data

This appendix documents the raw run-times used in the slowdown graphs. All run-times are expressed in seconds. For the Split-C/AM programs an NPB, all results are on 32 nodes unless otherwise specified.

| $o$ $\mu$s | Radix | EM3D(r) | EM3D(w) | Sample | Barnes | P-ray | Mur$\varphi$ | Connect | NowSort | Radb |
|---|---|---|---|---|---|---|---|---|---|---|
| 2.9 | 13.7 | 229.7 | 88.6 | 24.6 | 77.8 | 23.5 | 67.6 | 2.3 | 127.2 | 7 |
| 3.9 | 16.1 | 257.3 | 108.9 | 30 | 81.5 | 25 | 69.1 | 2.3 | 126.2 | 7.3 |
| 4.9 | 18.4 | 278.2 | 128.6 | 34.4 | 86.2 | 25.8 | 70.2 | 2.3 | 129.6 | 7.4 |
| 6.9 | 22.9 | 323.9 | 167.4 | 43.7 | - | 25.6 | 72.6 | 2.4 | 126.3 | 7.3 |
| 7.9 | 25.1 | 348.2 | 186.6 | 48.2 | 98.2 | 29.7 | 73.5 | 2.4 | 127 | 7.3 |
| 12.9 | 36.6 | 466.1 | 283.2 | 71.6 | 756.3 | 35.3 | 79.8 | 2.5 | 130.9 | 7.3 |
| 22.9 | 60.3 | 709.3 | 478.4 | 118 |  | - | 93.1 | 2.8 | 126.6 | 7.4 |
| 52.9 | 129.6 | 1439.8 | 1061.4 | 256.9 | - | 86.3 | 131.5 | 3.5 | 128.5 | 8 |
| 102.9 | 251 | 3047.4 | 2029.1 | 493.3 | - | 148.7 | 195.7 | 4.7 | 127.8 | 9.1 |

Table B.1: **Split-C/AM Run Times varying Overhead on 16 nodes**
*This table shows the run time, in seconds, of the Split-C/AM applications while varying the overhead on 16 nodes. This is the only result for these applications not run on 32 nodes.*

| $o$ $\mu$s | Radix | EM3D(r) | EM3D(w) | Sample | Barnes | P-ray | Mur$\varphi$ | Connect | NowSort | Radb |
|---|---|---|---|---|---|---|---|---|---|---|
| 2.9 | 7.8 | 114 | 38 | 13.2 | 43.2 | 17.9 | 35.3 | 1.2 | 56.9 | 3.7 |
| 3.9 | 10.5 | 138.7 | 48.1 | 16.1 | 50.1 | 19 | 37.1 | 1.2 | 56.7 | 3.8 |
| 4.9 | 13.2 | 161.6 | 58.1 | 18.7 | - | 19.6 | 37.7 | 1.2 | 61.2 | 3.8 |
| 6.9 | 18.7 | 208.8 | 77.4 | 23.8 | - | 22 | 41.8 | 1.2 | 57.9 | 3.8 |
| 7.9 | 21.5 | 232.9 | 87.4 | 26.5 | - | 20.8 | 41.9 | 1.2 | 58.3 | 3.8 |
| 12.9 | 36.3 | 354.4 | 138.5 | 39.3 | - | 28.2 | 46.2 | 1.3 | 58.1 | 3.9 |
| 22.9 | 68.9 | 600.1 | 236.2 | 65.2 | - | 39 | 51.2 | 1.3 | 58.3 | 4.1 |
| 52.9 | 198.2 | 1332.5 | 535.9 | 142.7 | - | 69.7 | 72.6 | 1.6 | 61.7 | 4.8 |
| 102.9 | 443.2 | 2551.7 | 1027.8 | 272.1 | - | 114 | 107.8 | 2.1 | 71.1 | 6.2 |

Table B.2: **Split-C/AM Run Times varying Overhead on 32 nodes**
*This table shows the run time, in seconds, of the Split-C/AM applications while varying the overhead on 32 nodes.*

| $g$ $\mu$s | Radix | EM3D(r) | EM3D(w) | Sample | Barnes | P-ray | Mur$\varphi$ | Connect | NowSort | Radb |
|---|---|---|---|---|---|---|---|---|---|---|
| 5.8 | 7.8 | 114 | 38 | 13.2 | 43.2 | 17.9 | 35.3 | 1.2 | 56.9 | 3.7 |
| 8.3 | 10.2 | 119 | 46.1 | 14.8 | 44.1 | 18.1 | 37.4 | 1.2 | 57.9 | 3.8 |
| 10.8 | 13 | 129.7 | 56.5 | 17.5 | 50.2 | 17.8 | 36.1 | 1.2 | 57.6 | 3.8 |
| 15.8 | 19.2 | 164.7 | 78.5 | 24.2 | 55.3 | 17.9 | 36.2 | 1.2 | 60.9 | 3.8 |
| 30.8 | 38.1 | 289.3 | 150.3 | 42.9 | 61.6 | 19.1 | 38.4 | 1.3 | 57.3 | 3.9 |
| 55.8 | 69.9 | 523 | 273.1 | 75.1 | 99.1 | 23.2 | 37.5 | 1.5 | 57.2 | 4 |
| 80.8 | 101.9 | 756.9 | 394 | 107.5 | 157.3 | 29 | 39.3 | 1.7 | 56.9 | 4.1 |
| 105.8 | 133.8 | 993.1 | 515.6 | 139.7 | 207.9 | 35.5 | 39.9 | 1.9 | 57.4 | 4.3 |

Table B.3: **Split-C/AM Run Times varying gap**
*This table shows the run time, in seconds, of the Split-C/AM applications while varying the gap.*

| $l$ $\mu$s | Radix | EM3D(r) | EM3D(w) | Sample | Barnes | P-ray | Mur$\varphi$ | Connect | NowSort | Radb |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7.8 | 114 | 38 | 13.2 | 43.2 | 17.9 | 35.3 | 1.2 | 56.9 | 3.7 |
| 7.5 | 8.5 | 144 | 39.4 | 13.3 | 50.7 | 19 | 39.6 | 1.2 | 57.6 | 3.8 |
| 10 | 8.5 | 159.8 | 39.7 | 13.3 | 48.6 | 20.1 | 36.9 | 1.2 | 56.5 | 3.8 |
| 15 | 8.5 | 198.7 | 41.2 | 13.3 | 57.6 | 22.5 | 36.4 | 1.2 | 57.1 | 3.8 |
| 30 | 10.2 | 320.8 | 45.9 | 13.3 | 70.5 | 28.2 | 39 | 1.3 | 57.2 | 3.9 |
| 55 | 10.7 | 523.1 | 56.4 | 13.3 | 103.6 | 39.9 | 39 | 1.4 | 57.5 | 3.9 |
| 80 | 11.1 | 726.6 | 70.3 | 13.3 | 131.6 | 49.9 | 36.4 | 1.6 | 60.2 | 4 |
| 105 | 12.2 | 943.5 | 87.1 | 13.3 | 162.3 | 61.4 | 38 | 1.7 | 61.5 | 4 |

Table B.4: **Split-C/AM Run Times varying Latency**
*This table shows the run time, in seconds, of the Split-C/AM applications while varying the latency.*

| Bandwidth MB/s | Radix | EM3D(r) | EM3D(w) | Sample | Barnes | P-ray | Mur$\varphi$ | Connect | NowSort | Radb |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.19 | 11.1 | 115.4 | 38.9 | 13.3 | 80.7 | 36.2 | 39.4 | 1.2 | 123.6 | 8.2 |
| 4.6 | 8.6 | 114.1 | 38.5 | 13.3 | 56.3 | 21.1 | 39.4 | 1.2 | 65.1 | 4.7 |
| 8.7 | 8.1 | 114.1 | 37.9 | 13.3 | 52.5 | 19.5 | 36.3 | 1.2 | 63.7 | 4.2 |
| 11.2 | 8.1 | 113.9 | 37.9 | 13.2 | 51.8 | 19.3 | 39 | 1.2 | 61.7 | 4 |
| 15 | 7.9 | 113.9 | 39.3 | 13.4 | 47.6 | 19.5 | 39.6 | 1.2 | 57.8 | 3.9 |
| 19 | 7.9 | 114 | 38.1 | 13.3 | 47.2 | 18.9 | 36.5 | 1.2 | 57.5 | 3.9 |
| 31 | 7.9 | 116 | 38.4 | 13.3 | 47.3 | 19.6 | 36.3 | 1.2 | 56.8 | 3.8 |
| 37.5 | 7.8 | 114 | 38 | 13.2 | 43.2 | 17.9 | 35.3 | 1.2 | 56.9 | 3.7 |

Table B.5: **Split-C/AM Run Times varying Bandwidth**
*This table shows the run time, in seconds, of the Split-C/AM applications while varying the Gap.*

| $o$ $\mu$s | FT | IS | MG |
|---|---|---|---|
| 10 | 173.2 | 18.7 | 17.8 |
| 11 | 178.1 | 21.3 | 18.3 |
| 20 | 175.8 | 22.5 | 19.9 |
| 30 | 177 | - | 23.9 |
| 60 | 184.9 | 22.5 | 26.6 |
| 110 | 222.1 | 33 | 26.6 |

Table B.6: **NPB Run Times varying Overhead**
*This table shows the run time, in seconds, of the NPB while varying overhead.*

| $g$ $\mu$s | FT | IS | MG |
|---|---|---|---|
| 5.8 | 173.2 | 18.7 | 17.8 |
| 8.3 | - | 21.3 | 18.2 |
| 10.8 | - | 21.5 | 16.8 |
| 15.8 | - | 20.1 | 16.8 |
| 30.8 | - | 19.6 | 18.3 |
| 55.8 | 170.6 | 21.3 | 22.6 |
| 80.8 | 159.1 | 21.1 | 21.3 |
| 105.8 | 171.6 | 21.4 | 23.5 |

Table B.7: **NPB Run Times varying gap**
*This table shows the run time, in seconds, of the NPB while varying gap. A switch bug caused* **FT** *to crash often. Unfortunately, the fixed switches were not available in time for this thesis, and the work-around would significantly perturb the results.*

| $l$ $\mu$s | FT | IS | MG |
|---|---|---|---|
| 5 | 173.2 | 18.7 | 17.8 |
| 7.5 | 178.4 | 22.6 | 18.2 |
| 10 | - | 20.6 | 19.2 |
| 15 | - | 21.1 | 17.8 |
| 30 | - | 21 | |
| 55 | - | 19.8 | 22.1 |
| 80 | - | 19.8 | 21.9 |
| 105 | - | 19.8 | 22.2 |

Table B.8: **NPB Run Times varying latency**
*This table shows the run time, in seconds, of the NPB while varying latency. A switch bug caused* **FT** *to crash for high latency. Unfortunately, the fixed switches were not available in time for this thesis, and the work-around would significantly perturb the results.*

| Bandwidth MB/s | FT | IS | MG |
|---|---|---|---|
| 1.19 | 661 | - | 60.9 |
| 4.6 | 225.7 | 22.6 | 25.5 |
| 11.2 | 186.2 | 23 | 20 |
| 15 | 171.8 | 22.1 | 19.6 |
| 19 | 177.1 | 22.1 | 23.5 |
| 37.5 | 173.2 | 18.7 | 17.8 |

Table B.9: **NPB Run Times varying Bandwidth**
*This table shows the run time, in seconds, of the NPB while varying Gap. Bandwidth is used so as to make an easier comparison to known systems.*

| Ops/sec | Latency ($\mu$s) | | | | | | |
|---|---|---|---|---|---|---|---|
| Measured | 10 | 50 | 100 | 500 | 1000 | 2000 | 4000 |
| 198 | 7.3 | 7.2 | 7.3 | 8.2 | 9.4 | 11.9 | 16.4 |
| 199 | - | - | - | - | - | 11.3 | - |
| 396 | - | 9.9 | - | - | - | - | - |
| 397 | 10.1 | - | 10 | 11 | 12.3 | 14.9 | - |
| 398 | - | - | - | - | - | - | 20.5 |
| 602 | 13.8 | 13.4 | 14 | 14.8 | - | - | 26.5 |
| 603 | - | - | - | - | 16.7 | 18.8 | - |
| 762 | - | - | - | - | - | - | 35.1 |
| 773 | - | - | - | - | - | - | 34.7 |
| 774 | - | - | - | - | - | - | 33.7 |
| 790 | - | - | - | - | - | - | 33.9 |
| 799 | - | 16.1 | 15.9 | 16.9 | - | 22 | - |
| 800 | 15.6 | - | - | - | - | - | - |
| 801 | - | - | - | - | 18.8 | - | - |
| 943 | - | - | - | - | - | 28.8 | - |
| 949 | - | - | - | - | - | 25.6 | - |
| 950 | 17.6 | - | - | 18.8 | 21.8 | - | - |
| 951 | - | - | 18.3 | - | - | - | - |
| 952 | - | 17.3 | - | - | - | - | - |
| 958 | - | - | - | - | - | 27.7 | - |
| 1000 | - | - | - | - | 24.9 | - | - |
| 1006 | 18.9 | - | 21.1 | 20.1 | - | - | - |
| 1007 | - | 19 | - | - | - | - | - |
| 1009 | - | - | - | - | 25.8 | - | - |
| 1045 | - | - | 23.8 | - | - | - | - |
| 1053 | 19.9 | - | - | - | - | - | - |
| 1054 | - | 20 | - | - | - | - | - |
| 1055 | - | - | - | 21.5 | - | - | - |

Table B.10: **SPECsfs Response Times in varying Latency on the SCSI system**
*The table shows the average response time, in milliseconds, for the SCSI system running NFS version 2 over UDP using the SFS 2.0 operation mix while varying latency.*

| Ops/sec | Latency ($\mu$s) | | | | | | |
|---|---|---|---|---|---|---|---|
| Measured | 10 | 15 | 50 | 100 | 150 | 1000 | 4000 |
| 298 | 3.4 | 3.5 | 3.8 | 3.7 | 3.9 | 5.9 | 13.4 |
| 397 | 4 | 3.9 | - | 4 | 4.1 | - | - |
| 398 | - | - | 4 | - | - | 6.3 | 16.4 |
| 494 | 4.5 | 4.5 | - | 4.6 | 4.9 | 7 | - |
| 495 | - | - | 4.6 | - | - | - | 20.5 |
| 595 | 4.9 | 5.2 | 4.9 | 5 | 5.2 | 7.6 | 17.4 |
| 696 | 6.1 | 5.8 | 6 | 6.2 | 6.2 | 8.7 | 20.1 |
| 791 | - | - | - | - | - | - | 24.5 |
| 797 | 7.2 | 7.2 | 7.3 | 7.6 | 7.9 | - | - |
| 798 | - | - | - | - | - | 10.6 | - |
| 900 | - | - | 9 | - | - | - | - |
| 901 | 8.8 | - | - | - | - | - | - |
| 902 | - | 9 | - | 9.5 | 9.4 | - | 25.8 |
| 903 | - | - | - | - | - | 12.8 | - |
| 1001 | 11.4 | - | - | - | - | - | - |
| 1002 | - | - | - | 11.9 | 11.1 | - | - |
| 1003 | - | 11.5 | 10.8 | - | - | - | - |
| 1004 | - | - | - | - | - | 13.7 | - |
| 1006 | - | - | - | - | - | - | 31.8 |
| 1069 | - | - | - | - | - | - | 46.3 |
| 1121 | - | - | - | - | - | - | 44.2 |
| 1170 | - | - | - | - | - | - | 42.3 |
| 1188 | - | - | - | - | - | - | 41.3 |
| 1204 | - | 14.8 | 14.6 | 14.6 | - | - | - |
| 1205 | 15 | - | - | - | - | - | - |
| 1207 | - | - | - | - | - | 17.1 | - |
| 1304 | 18.7 | - | - | - | - | - | - |
| 1305 | - | 18.2 | - | - | - | - | - |
| 1306 | - | - | 18.1 | 18.5 | - | - | - |
| 1307 | - | - | - | - | - | 22.3 | - |
| 1400 | - | 21.6 | 22.1 | 21.1 | - | - | - |
| 1402 | - | - | - | - | - | 25.7 | - |
| 1405 | 21 | - | - | - | - | - | - |
| 1499 | - | 31.2 | - | - | - | - | - |
| 1501 | - | - | - | - | - | 30.1 | - |
| 1503 | 30.8 | - | - | - | - | - | - |
| 1505 | - | - | - | 31 | - | - | - |

Table B.11: **SPECsfs Response Times varying Latency on the RAID system**
*The table shows the average response time, in milliseconds, for the RAID running NFS version 2 over UDP using the SFS 2.0 operation mix while varying latency.*

| Ops/sec | Overhead ($\mu$s) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Measured | 80 | 85 | 90 | 95 | 100 | 105 | 130 | 180 | 280 |
| 298 | 3.4 | - | - | 3.9 | 3.6 | 3.8 | 4.1 | 5.1 | 6.6 |
| 299 | - | 3.8 | - | - | - | - | - | - | - |
| 397 | 4 | - | - | 4 | 4.2 | - | - | - | - |
| 398 | - | 4 | - | - | - | 4.3 | 4.8 | 5.6 | 7.8 |
| 494 | 4.5 | 4.5 | - | 4.7 | 4.6 | 4.7 | 5.3 | 6.6 | 10.1 |
| 595 | 4.9 | 5 | - | 5.3 | 5.4 | 5.3 | 6.3 | - | 14.9 |
| 596 | - | - | - | - | - | - | - | 7.7 | - |
| 615 | - | - | - | - | - | - | - | - | 90.3 |
| 627 | - | - | - | - | - | - | - | - | 89.8 |
| 638 | - | - | - | - | - | - | - | - | 89.5 |
| 651 | - | - | - | - | - | - | - | - | 88.3 |
| 660 | - | - | - | - | - | - | - | - | 87.5 |
| 679 | - | - | - | - | - | - | - | - | 86.7 |
| 692 | - | - | - | - | - | - | - | - | 86.2 |
| 696 | 6.1 | 6 | - | 6 | 6.3 | - | 7.1 | 9.2 | - |
| 697 | - | - | - | - | - | 6.4 | - | - | - |
| 704 | - | - | - | - | - | - | - | - | 52.9 |
| 797 | 7.2 | 7.4 | - | 7.9 | 7.6 | 8 | 9 | 12.1 | - |
| 901 | 8.8 | 8.5 | - | 9.4 | - | 9.8 | - | - | - |
| 902 | - | - | - | - | 9 | - | - | - | - |
| 903 | - | - | - | - | - | - | 10.3 | 17.3 | - |
| 987 | - | - | - | - | - | - | - | 55.7 | - |
| 1001 | 11.4 | - | - | - | - | 12.8 | - | - | - |
| 1002 | - | - | - | 11.9 | 11.6 | - | 14.1 | - | - |
| 1003 | - | 11.5 | - | - | - | - | - | - | - |
| 1127 | - | - | - | - | - | - | 48.9 | - | - |
| 1168 | - | - | - | - | - | - | 47.9 | - | - |
| 1204 | - | 15.7 | - | - | - | - | - | - | - |
| 1205 | 15 | - | - | 16.2 | - | - | - | - | - |
| 1206 | - | - | - | - | - | 19 | - | - | - |
| 1207 | - | - | - | - | 17.6 | - | 24.6 | - | - |
| 1217 | - | - | - | - | - | - | 46.5 | - | - |
| 1282 | - | - | - | - | - | 41.8 | - | - | - |
| 1304 | 18.7 | - | - | - | - | 21.9 | - | - | - |
| 1307 | - | - | - | 22.7 | - | - | - | - | - |
| 1309 | - | 20.4 | - | - | 23.2 | - | - | - | - |
| 1330 | - | - | - | - | 40.6 | - | - | - | - |
| 1331 | - | - | - | - | - | 40.4 | - | - | - |
| 1338 | - | - | - | 31.4 | - | - | - | - | - |
| 1359 | - | - | - | 39 | - | - | - | - | - |
| 1378 | - | - | - | - | 39 | - | - | - | - |
| 1404 | - | 21.8 | - | - | - | - | - | - | - |
| 1405 | 21 | - | - | - | - | - | - | - | - |
| 1456 | - | 35.3 | - | - | - | - | - | - | - |
| 1503 | 30.8 | - | - | - | - | - | - | - | - |

Table B.12: **SPECsfs Response Times varying Overhead on the RAID**
*The table shows the average response time, in milliseconds, for the RAID running NFS version 2 over UDP using the SFS 2.0 operation mix while varying overhead.*

| Ops/sec | Overhead ($\mu$s) | | | | | | |
|---|---|---|---|---|---|---|---|
| Measured | 80 | 105 | 130 | 180 | 280 | 480 | 580 |
| 198 | 7.3 | 7.1 | 7.4 | 7.6 | 8.5 | 9.7 | 10.5 |
| 396 | - | - | - | 10.8 | - | - | - |
| 397 | 10.1 | 10.1 | 10.2 | - | 11.7 | - | - |
| 398 | - | - | - | - | - | 14.6 | 16.6 |
| 538 | - | - | - | - | - | - | 54.6 |
| 602 | 13.8 | 13.8 | 14 | - | - | - | - |
| 603 | - | - | - | 14.7 | 16.7 | - | - |
| 604 | - | - | - | - | - | 32.1 | - |
| 798 | - | - | 16.9 | 18.1 | - | - | - |
| 799 | - | 16.2 | - | - | 25.8 | - | - |
| 800 | 15.6 | - | - | - | - | - | - |
| 801 | - | - | - | - | 35.9 | - | - |
| 950 | 17.6 | 18.7 | - | - | - | - | - |
| 951 | - | - | - | 23.2 | - | - | - |
| 952 | - | - | 19.5 | - | - | - | - |
| 1005 | - | 21.8 | 21.5 | - | - | - | - |
| 1006 | 18.9 | - | - | - | - | - | - |
| 1029 | - | 24.9 | - | - | - | - | - |
| 1042 | - | - | 24.8 | - | - | - | - |
| 1053 | 19.9 | - | - | - | - | - | - |

Table B.13: **SPECsfs Response Times varying Overhead on the SCSI system**
*The table shows the average response time, in milliseconds, for the SCSI system running NFS version 2 over UDP using the SFS 2.0 operation mix while varying overhead.*

| Ops/sec | Bandwidth(MB/s) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Measured | 26 | 14.5 | 11.3 | 10.6 | 9.1 | 5.9 | 4.5 | 2.5 | 1.2 |
| 198 | 7.1 | 7.2 | 7.3 | 7.3 | 7.2 | 7.5 | 7.5 | 7.9 | 10.6 |
| 396 | 9.8 | 10 | 10 | - | - | 10.2 | - | 11.3 | - |
| 397 | - | - | - | - | 10.1 | - | 10.4 | - | 15.7 |
| 602 | - | 13.5 | - | - | 13.6 | - | - | - | - |
| 603 | - | - | 14 | - | - | 13.9 | 14.3 | 15.7 | - |
| 604 | - | - | - | - | - | - | - | - | 25.3 |
| 758 | - | - | - | - | - | - | - | - | 35.8 |
| 759 | - | - | - | - | - | - | - | - | 35.9 |
| 762 | - | - | - | - | - | - | - | - | 35.7 |
| 774 | - | - | - | - | - | - | - | - | 34.4 |
| 797 | - | - | 15.9 | - | 15.7 | 15.8 | 16.5 | - | - |
| 798 | 15.5 | 15.8 | - | - | - | - | - | - | - |
| 799 | - | - | - | - | - | - | - | 18.4 | - |
| 905 | 17 | - | - | - | - | - | - | - | - |
| 949 | - | - | 17.4 | - | - | 18.4 | - | 20.9 | - |
| 950 | - | - | - | - | 18.4 | - | 18.7 | - | - |
| 951 | - | 17.5 | - | - | - | - | - | - | - |
| 952 | 17.4 | - | - | - | - | - | - | - | - |
| 1004 | - | - | - | - | - | 20.1 | - | - | - |
| 1005 | - | - | 19.1 | - | 19.4 | - | 19.8 | - | - |
| 1006 | - | 19 | - | - | - | - | - | - | - |
| 1007 | 18.9 | - | - | - | - | - | - | 22.1 | - |
| 1014 | 27.7 | - | - | - | - | - | - | - | - |
| 1039 | 26.9 | - | - | - | - | - | - | - | - |
| 1052 | - | - | - | - | - | - | 21.4 | - | - |
| 1053 | - | - | - | - | 20.3 | - | - | 23.6 | - |
| 1054 | 20.3 | - | 20.5 | - | - | - | - | - | - |
| 1055 | - | 20 | - | - | - | 21.2 | - | - | - |
| 1074 | 26.1 | - | - | - | - | - | - | - | - |
| 1089 | 24.9 | - | - | - | - | - | - | - | - |
| 1097 | 22.7 | - | - | - | - | - | - | - | - |

Table B.14: **SPECsfs Response Times varying Bandwidth on the SCSI system**
*The table shows the average response time, in milliseconds, for the SCSI system running NFS version 2 over UDP using the SFS 2.0 operation mix while varying bandwidth.*