Improving the Performance of Shared Virtual Memory on System Area Networks

Angelos Bilas

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 1998

© Copyright by Angelos Bilas, 1998.
All Rights Reserved

Abstract

As clusters of workstations, uniprocessor or symmetric multiprocessors (SMPs), become important platforms for parallel computing, there is increasing research interest in supporting the attractive, shared address space programming model across them in software. The reason is that it may provide successful low–cost, high–performance alternatives to both tightly–coupled, hardware–coherent distributed shared memory machines and to scalable servers. In both these cases, the clusters are formed with off–the–self, high–end PCs or workstations and system area networks that track technologies well. Given that a shared memory abstraction is an attractive programming model for this architecture, there has been a lot of research in fast communication on clusters connected with system area networks and in protocols for supporting software shared memory across them. However, the end performance of applications that were written for the more proven hardware–coherent shared memory is still not very good on these systems.

This dissertation is about improving the performance of shared virtual memory on clusters that are connected with system area networks. In this architecture, three major layers of software (and hardware) stand between the end user and performance, each with its own functionality and performance characteristics. The lowest layer is the *communication layer*: the communication hardware and the low level software libraries that provide basic messaging facilities. Next is the *protocol* layer that provides the programming model to the parallel application programmer. Finally, above the programming model or protocol layer runs the *application* itself. This dissertation improves the performance of shared virtual memory on clusters by studying and improving each of this layers and some of their interactions.

Acknowledgments

...as she kept on walking she turned her head back toward him, smiled, and lifted her right arm in the air, easily, flowingly, as if she were tossing a brightly colored ball. That instant ... was miraculous. ...It was an unforgettable moment... That gesture was so unexpected and beautiful that it remained in [Agnes's] memory like the imprint of a lightning bolt... — Immortality, Milan Kundera.

Maybe it is moments like this one that remain. And the Princeton years gave me plenty of those.

I would like to thank the people that surrounded me in the department all these years and contributed to a, to say the least, pleasant and productive environment. The faculty members of the Computer Science department, and especially Douglas Clark, Edward Felten, Kai Li and my advisor Jaswinder Pal Singh for their guidance and encouragement. I owe a lot particularly to Kai for his guidance throughout this effort and most of all to my advisor J.P., whose assistance made this work possible. His guidance and support in the course of this work all these years have been invaluable.

Cezary Dubnicki has played multiple roles during the course of this work; working closely with him has been a most rewarding experience. I would also like to thank Richard Alpert, Mathias Blumrich, Yuqun Chen, Stefanos Damianakis, Sanjeev Kumar, Yuanyuan Zhou, and in particular Liviu Iftode and Rudrajit Samanta, for the useful discussions during the course of this work. Also, Dongming Jiang and Hongzhang Shan for their help with many of the applications, and NEC Research and particularly James Philbin and Jan Edler for providing simulation cycles and a stable environment for parts of this work.

I am thankful to CS staff for providing their technical expertise on many issues and for their patience with my requests. Similarly, I owe thanks to Melissa, Sandy, and Trish, for providing a carefree environment, where most issues a student does not know how to deal with, were already taken care of, and for providing a devastatingly non–technical and precise rephrasing for every technical problem. I would also like to thank our funding agencies, and mention that this work was supported by DARPA grant under contract N00014-95-1-1144, NSF grant MIP-9420653, NSF grant CDA96-24099, and Intel Corporation.

Intensive periods always constitute a test for relations of all kinds, and raise the opportunity for new ones to be created. It is difficult to find words to thank George Apostolopoulos for being there in good and bad. George was loaded with the heavy burden of knowing everything that happened. Close friends since we started our studies back in Greece, our friendship continued and evolved, providing support all these years.

Alexander Michaelides, for starting this trip together, and discovering its difficulties, frustrations and great moments. Dimitrios Kyritsis for joining along. Dimitrios has been a delight to converse with, bearing the interesting property that every conversation with him turned into a debate with almost no point of convergence. Claire Adjiman, Dawn Kataoka, and Omar Matar, for being there whenever a break was needed. Evdokia for being bigger than life.

Panagiotis ktorides who was always able to perceive things without me having to say much. Zoe for showing me that circumstances do not matter. Christos Tzomakas with whom friendship continued, although our quest for the "new" kept us in different continents. Nick Koudas for his support in the first year and especially that Tuesday night back in the November of the first semester. Paola for bringing me closer to new frontiers.

The Program in Hellenic Studies, Dimitrios Gondicas, Dimitrios Gunopoulos, George Vetoulis, Andromache Karanika, Anastasion Viglas and the rest of the members of the Hellenic Association for contributing to maintaining a link with “home”. My officemates, Annalisa, Peter, Scott, and Lisa, that enriched the long hours in the office with interesting, real–world discussions.

I am not sure I know how to thank my uncle Christos, my aunt Erofili, and my cousins Despina and Michael for making the short visits home as rich as they can be, wanting me to taste as much as possible of what I could have missed. But I am sure that they are the closest thing to parents, brother, and sister that can ever be.

Finally, I know by now, that I owe more than I can comprehend to my family. My grandmother Aggeliki, whose sole concern, while with us, was for me and my brother to study. She used to say “my child’s children are twice my children”. I wonder if she knew that “my mother’s mother is twice my mother”. My brother George, whose attitude of life, “different” and unconventional, has provided an alternative view of the world, any world, and has been a source of both strength and calmness. My parents Apostolos and Aikaterini, who gave us more than we could imagine. Their dream has always been to see us reach our dreams. And they sacrificed everything for it. They made possible this colorful journey from a primary school in my hometown, Katerini, in Greece, to this thesis in Princeton, and who knows where else...

Στη μνήμη της γιαγιάς μου Αγγελικής

Στους γονείς μου Απόστολο και Αικατερίνη

Στον αδερφό μου Γιώργο

To the memory of my grandmother Aggeliki

To my parents Apostolos and Aikaterini

To my brother George

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As clusters of symmetric multiprocessors become important platforms for parallel computing, there is increasing research interest in supporting the attractive, shared address space programming model across them in software. The traditional reason is that this approach may provide low–cost alternatives to tightly–coupled, hardware–coherent Distributed Shared Memory (DSM) machines. A more important reason, however, is that clusters and hardware DSM machines are emerging as the two major types of platforms available to users of multiprocessing, who would like to write parallel programs once and run them on both types of platforms.

Moreover, clusters constitute an alternative to scalable servers. Low cost and high performance are the potential advantages compared to existing solutions that motivate building a high–performance server out of a network of commodity computer systems. Such servers can cost substantially less than custom–designed ones because they can leverage high–volume commodity hardware and software. They can also deliver high performance because commodity hardware and software track technologies well.

In both the above cases, the clusters are formed with off–the–shelf components, high–end PCs or workstations (uniprocessors or SMPs) and system area networks. Given that a shared memory abstraction is an attractive programming model for this architecture, a lot of research has been done in fast communication on clusters of uniprocessor systems connected with system area networks and in protocols for supporting software shared memory across them. However, the end performance of applications that were written for the more proven hardware–coherent shared memory is still not very good on these systems.

This dissertation is about shared virtual memory (SVM) [59] on clusters of symmetric multiprocessors (SMPs) that are connected with system area networks (SANs). In this architecture, three major layers (Figure 1.1) of software (and hardware) stand between the end user and performance, each with its own functionality and performance characteristics. The lowest layer is the *communication layer*: the communication hardware and the low–level software libraries that provide basic messaging facilities. Next is the *protocol* layer that provides the programming model to the parallel application programmer. Finally, above the programming model or protocol layer runs the *application* itself. Work reported in this dissertation improves the performance of shared virtual memory on clusters by studying the two (lower) system layers and their interactions.

This layered view of the system provides a useful framework for improving system performance and for presenting the results; the presentation of this work follows the layers in Figure 1.1 in a bottom–up fashion.

1.1 Efficient and Reliable Communication

Chapter 2 is concerned with the communication layer itself, and is not specialized for SVM. A lot of research has been conducted in providing fast interconnection networks

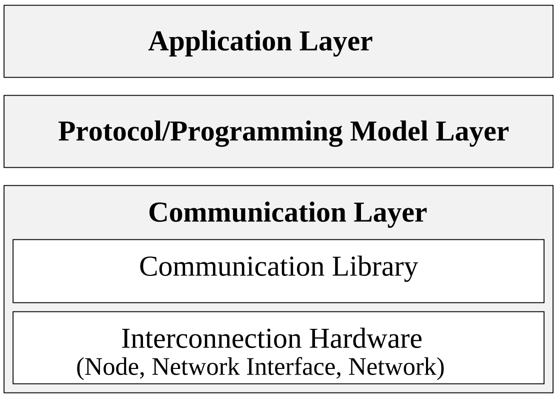

Figure 1.1: The layers that affect the end application performance in software shared memory.

for parallel systems. Previous work [95, 94, 4, 74] found that significant overheads in communicating among nodes are due to the interface between the host and the network and not the network interconnect itself. For this reason, recent efforts have mostly focused in improving the network interfaces that connect the nodes to the network interconnect, rather than the interconnect itself.

The most important feature of the network interface is the level of functionality provided to the user, since it affects most of the issues in the network interface design space. Higher functionality requires tighter integration of the network interface in the system. Thus support for cache coherence and load/store APIs leads to integrating the network interface on the memory bus, or even closer to the processor. More standard message passing APIs with explicit communication operations can be implemented at a lower level of integration on the I/O bus.

The level of integration of the network interface into the node has many effects on the system: (i) the tighter the integration, the higher the cost of the network interface, (ii) more importantly, the tighter the integration the longer the design cycle and the design cost. Although the absolute performance of a system has traditionally depended significantly on the level of integration, it is not clear whether loosely coupled systems must exhibit inherently worse performance than tightly coupled systems. Thus, since loosely coupled systems follow the technology curves much better, if they are demonstrated to perform reasonably well, they can provide superior cost–performance ratios.

For these reasons, it is very important to investigate whether network interfaces that are loosely coupled with the memory subsystem and the processor can deliver high performance to the end user. Moreover, in certain cases [15], hybrid solutions that cover the space between loosely and tightly coupled network interfaces have been proposed. These network interfaces are usually located on the I/O bus and require support from other parts of the system (e.g., the memory bus) that is less intrusive than the support required by more tightly integrated network interfaces.

Research efforts to provide fast, loosely–coupled multicomputer interconnection networks for clusters of workstations at low cost have led to the design of network interfaces that implement minimal functionality in hardware and use powerful basic mechanisms that allow communication system designers to implement in software the functionality needed by different paradigms and systems using these basic mechanisms. These network interfaces may encompass a general purpose processor, they use simple switches to route packets to their destination, and they allow for arbitrary network topologies. For design simplicity and flexibility, and cost reasons, these characteristics result in the absence of reliability in the network interface hardware and the need to support routing for arbitrary, dynamic topologies in the communication system. At the software layer, the common framework is to implement a user–level data transport protocol that integrates well with the network interface. Then, it is necessary to see if existing high–level protocols for shared virtual memory and message–passing can be implemented efficiently. The goal is to deliver to the user communication performance close to the hardware's limit while providing the full functionality of the high–level APIs.

In this work we address of the issues related to the design of the communication layer: (i) We show that low–latency, high–bandwidth communication can be provided with off–the–shelf components. (ii) We propose incorporating reliable communication in the network interface firmware at very low cost. (iii) We propose a new scheme for dynamically determining the network topology; this mechanism relies on the retransmission mechanism that is used to provide reliable communication.

If performance is a high priority goal, these problems are highly non–trivial and many issues need to be considered. Data copies and unnecessary overheads need to be avoided both in the host and the network interface, algorithms and protocols need to be simple and memory requirements must be low, since the network interface has limitations in both processing speed and memory.

We demonstrate the ideas by presenting an efficient, reliable and flexible communication layer for System Area Networks (SANs). More specifically we describe the design and implementation of the virtual memory–mapped communication model (VMMC) on a Myrinet [17] network of PCI–based [70] PCs; similar approaches can be taken in other communication layers as well. VMMC is a communication model, which provides direct data transfer between the sender's and receiver's virtual address spaces. It also eliminates operating system involvement in communication, provides full protection, supports user–level buffer management and zero–copy protocols, and minimizes software communication overhead.

The implementation of the basic VMMC mechanism on Myrinet achieves about 11 μs one–way latency and provides about 100 MBytes/s user–to–user bandwidth. This demonstrates that low–latency, high–bandwidth communication can be provided with off–the–shelf hardware components. Moreover the cost of reliable communication is minimal: less than 2.5μs for latency and less than 10% for all different types of bandwidth. Similarly, since the support for dynamic system configuration is not on the common path, there is no effect on system performance when no changes in the system topology occur.

1.2 SVM for networks of SMPs

As SMPs become increasingly widespread, due to performance, economical, and software and hardware design flexibility reasons, it is very attractive to build larger multiprocessors by putting together SMP nodes rather than uniprocessor nodes. Commodity network interfaces and networks have progressed to the point where relatively low latency and high bandwidth are achievable, making such clusters of SMPs all the more attractive.

The first key question is what programming model to use across nodes. The choices are to extend the coherent shared address space abstraction that is available within the nodes, or to use a shared address space within nodes and explicit message passing between nodes, or to use explicit message passing everywhere by using the hardware–supported shared memory within a node only to accelerate message passing, not to share data among processors. A coherent shared address space has been found to be an attractive programming model: it offers substantial ease of programming over message passing for a wide range of applications—especially as applications become increasingly complex and irregular as we try to solve more realistic problems—and it has also been shown to deliver very good performance when supported in hardware in tightly coupled multiprocessors, at least up to the 64-128 processor scale where experiments have been performed [58, 57, 48, 87, 88, 96, 89]. It is also the programming model of choice for small–scale multiprocessors, so provides a graceful migration path. The last of the programming model possibilities (message passing everywhere), not only forces programmers to use explicit message passing but also does not take full advantage of hardware coherence within the SMP, and the second one provides an awkward hybrid model that is unattractive to programmers.

Unfortunately, commodity SMP nodes and networks do not provide hardware support for a coherent shared address space *across* nodes. Shared virtual memory is a method of providing coherent replication in a shared address space across nodes without specialized hardware support beyond that already available in uniprocessors. The idea is to provide the replication and coherence in main memory through the virtual memory system, so main memory is managed as a cache at page granularity. The coherence protocol runs in software, and is invoked on a page fault, just as a hardware cache coherence protocol is invoked on a cache miss. A problem with page–level coherence is that it causes a lot of false sharing when two unrelated items that are accessed by different processors (and written by at least one of them) happen to fall on the same page. Since protocol operations and communication are expensive, this false sharing is particularly harmful to performance. To alleviate the effects of false sharing, protocols based on relaxed memory consistency models have been developed [35, 52, 6, 45]. These models allow coherence information to be propagated only at synchronization points rather than whenever shared data are modified. This means that if one processor is repeatedly writing a word on a page and another processor is repeatedly reading another unrelated word on the same page, they can keep doing this independently until they reach synchronization points, at which time the pages are made consistent. To also allow multiple writers to the same page to write their separate copies independently until a synchronization point, so called multiple–writer protocols have been developed. These greatly alleviate the effects of false sharing, but communication and the propagation of coherence information are still expensive when they do occur. Much research has been done in this area, and many good protocols have been developed [51, 52, 19, 49, 29, 77, 44, 45, 100].

One way to provide the shared memory programming model in clusters is to extend these SVM protocols to use multiprocessor (SMP) rather than uniprocessor nodes. Another view of this approach is that the less efficient SVM is used not as the basic mechanism with which to build multiprocessors out of uniprocessors, but as a mechanism to extend available small–scale machines to build larger machines while preserving the same desirable programming model. The key is to use the hardware coherence support available within the SMP nodes as much as possible, and resort to the more costly SVM protocol across nodes only when necessary. If successful, this approach can make a coherent shared address space a viable programming model for both tightly–coupled multiprocessors (using hardware cache coherence) and loosely coupled clusters.

Clustering processors together using a faster and finer–grained communication mechanism has some obvious advantages, namely prefetching, cache–to–cache sharing, and overlapping working sets [28]. The hope is that for many applications a significant fraction of the interprocessor communication may be contained within each SMP node. This reduces the amount of expensive (high latency and overhead) cross–node SVM communication needed during the application's execution. However, it unfortunately *increases* the bandwidth (i.e., communication per–unit time) demands on the node–to–network interface. This is because the combined computation power within the SMP node typically increases much faster with cluster size c (linearly) than the degree to which the per–processor cross–node communication volume is reduced. This means that depending on the constants, the node–to–network bandwidth may become a bottleneck if it is not increased considerably when going from a uniprocessor to an SMP node.

A recent and promising form of SVM protocols is the class of so–called *home–based* protocols (HLRC, AURC,Cashmere) [44, 45, 46, 100, 54, 91]. Chapter 3 describes a protocol for home–based SVM across SMP nodes (HLRC-SMP, AURC-SMP) that accomplishes the goals above. The SVM protocol can operate completely in software, or can exploit hardware support for *automatic update (AU)* propagation of writes to remote memories (also called automatic write propagation) as supported in the SHRIMP multicomputer [16], and in a different way in the DEC Memory Channel [37]. Having described the protocol, we use detailed simulation to examine how using k, c–processor SMPs connected this way compares in performance to using SVM across $k * c$ uniprocessor nodes, and whether the performance characteristics look promising overall. We identify where time is spent in real applications and hence where key system bottlenecks are.

A key question for such systems is how much and what kind of limited hardware support is most effective in accelerating their performance by alleviating the observed bottlenecks, bringing it closer to that of full hardware coherence and making the shared address space model attractive for application users on clusters as well. The most popular form of hardware support used so far is the propagation of fine-grained writes to remote memories [16, 50, 37].

Previous work [44, 46] has shown that the home–based protocols with hardware support for automatic update (AURC) and without (HLRC) outperform previous all–software protocols. Also it demonstrated that support for automatic update is beneficial in systems with uniprocessor nodes and customized network interfaces.

Snooping–based automatic update has relied on write–through cache hierarchies so shared writes appear on the memory bus. However, the current generation of microprocessor based systems, including Intel–based PCs and especially SMPs, do not support write–through second–level caches. To use AU in SVM protocols, we must design new mechanisms that work with write–back caches. Another important development is the appearance of efficient commodity NIs such as Myrinet [17] in the marketplace. How beneficial AU support will be with these NIs (which will still require snooping support for AU) is unclear. Finally, an important trend is the use of commodity SMPs rather than uniprocessors as the building blocks for clusters, which can improve software shared memory performance [19, 49, 9, 77].

This work answers the the following questions: (i) Given the decline of write–through capable second–level caches—and the traffic problems they raise with SMP nodes—can hardware automatic update support be provided even with write–back caches? (ii) Within the same (home–based) protocol framework, does hardware support for automatic update snooping deliver a performance benefit large enough to justify its cost, assuming both write–through capable and write–back caches? (iii) Does this hardware support for AU work well enough with the new commodity NIs like Myrinet, or does it require customized NIs like that in SHRIMP to achieve substantial benefits? That is, is AU–based SVM an argument for building customized NIs?

The answers are given in the context of a home–based approach in the protocol layer in all cases, providing a consistent framework; similar approaches can be taken for other protocols as well. We address the above questions through detailed simulation and for a wide range of real applications and computational kernels. The simulator gives us deep visibility into all aspects of the simulated hardware and software, including queues and buffers, which is especially important since the focus here is on the communication architecture. An associated flexible visualization tool allows us to easily examine the detailed event frequencies and contention–related bottlenecks in all parts of the node and NI on a per–node basis, and proves very important in analyzing the observed performance effects.

We find that: (i) the performance of both protocols (AURC and HLRC) improves substantially with the use of SMP rather than uniprocessor nodes in five of ten applications programs. In three applications there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ across all–software and automatic update protocols, with the latter performing worse with SMPs than with uniprocessors; (ii) AU support with write–through caches does significantly improve performance of some irregular applications, especially when diff–related costs dilate critical sections and increase serialization, but it can also hurt substantially in some cases; (iii) AU support with write–back caches solves the problems caused by the increased traffic and performs best, but is very intrusive into the underlying node; (iv) overall, the benefits are quite limited with commodity NIs with their higher occupancies per packet, and NIs customized for AU packet generation may be important for using AU effectively.

1.3 SVM perfromance Bottlenecks

The end application performance of an SVM system, or of any implementation of a programming model, relies on the performance and interactions of the three layers that sit between the problem to be solved and the hardware (Figure 1.1): the application or program layer, the protocol layer that supports the programming model, and the communication layer that implements the machine's communication architecture. Each of the system layers has both performance and functionality characteristics, which can be enhanced to improve application performance.

In the last few years there has been much improvement of SVM protocols and systems, and several applications have been restructured to improve performance [51, 44, 45, 100, 54]. With this progress, it is now interesting to examine what are the important *system* bottlenecks that stand in the way of effective parallel performance;

in particular, which parameters of the communication architecture are most important to improve further relative to processor speed, which are already adequate on modern systems for most applications, and how will this change with technology in the future. Such studies can hopefully assist system designers in determining where to focus their energies in improving performance, and users in determining what system characteristics are appropriate for their applications.

Chapter 4 examines these questions through detailed architectural simulation using applications with widely different behavior. We simulate a cluster architecture with SMP nodes and a fast system area interconnect with a programmable network interface (Myrinet). We use the *home–based* SVM protocol of Chapter 3. This protocol has been demonstrated to have comparable or better performance than other families of SVM protocols. The base case of the protocol, called home–based lazy release consistency (HLRC-SMP) does not require any additional hardware support. We later examine a variant (AURC-SMP), that uses automatic (hardware) propagation of writes to remote nodes to perform updates to shared data, and also extend our analysis to the use of uniprocessor nodes where this is useful. The major performance parameters we consider are the host processor *overhead* to send a message, the network interface *occupancy* to prepare and transfer a packet, the node–to–network *bandwidth* (often limited by I/O bus bandwidth), and the *interrupt cost.* We do not consider network link latency, since it is a small and usually constant part of the end–to–end latency in system area networks (SAN). After dealing with performance parameters, we also briefly examine the impact of key granularity parameters of the communication architecture. These are the page size, which is the granularity of coherence, and the number of processors per node.

Assuming a realistic system that can be quite easily implemented today, and a range of applications that are well optimized for SVM systems [47], we see (Figure 1.2)

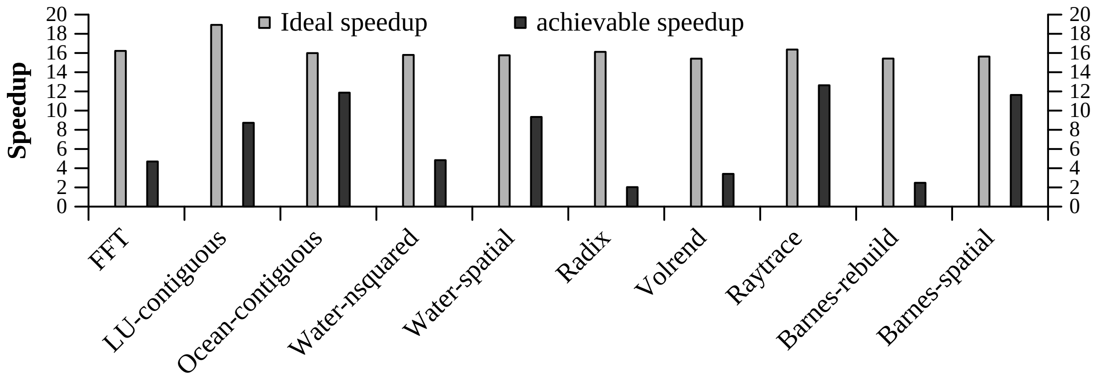

Figure 1.2: Ideal and realistic speedups for each application. The ideal speedup is computed as the ratio of the uniprocessor execution time divided by the sum of the compute and the local cache stall time in the parallel execution, i.e., ignoring all communication and synchronization costs. The realistic speedup corresponds to a realistic set of values for communication architecture parameters today, in a configuration with four processors per node.

that, for most applications, protocol and communication overheads are substantial. The speedups obtained in the realistic implementation are much lower than in the ideal case, where all communication costs are zero. This motivates the current research, whose goal is twofold. First, we want to understand how performance changes as the parameters of the communication architecture are varied relative to processor speed, both to see where we should invest systems development energy and to understand the likely evolution of system performance as technology evolves. For this, we use a wide range of values for each parameter and we focus on three specific points in the parameter space. The first is the point for which the system generally achieves its best performance within the ranges of parameter values we examine. The performance on an application at this point is called its **best** performance. The second point is an aggressive set of values that the communication parameters can have in current or near–future systems, especially if certain operating system features are well optimized. The performance at this point in the space is called the **achievable** performance. The third point is the **ideal** point, which represents a hypothetical system that incurs no communication or synchronization overheads, taking into consideration only compute time and stall time on local data accesses. Our goal is to understand the gaps between the achievable, best and ideal performance by identifying which parameters contribute most to the performance differences. This leads us to the primary communication parameters that need to be improved to close the gaps.

Our results show the relative effect of each of the parameters, i.e. relative to the processor speed and to one another. While the absolute parameter values that are used for the achievable set match what we consider achievable with aggressive current or near–future systems, viewing the parameters relative to processor speed allows us to understand what the behavior will be as technology trends evolve. For instance, if the ratio of bandwidth to processor speed changes, we can use these results to reason about system performance.

We find, somewhat surprisingly, that host overhead to send messages and per–packet network interface occupancy are not critical to application performance. In most cases, interrupt cost is by far the dominant performance bottleneck, even though our protocol is designed to reduce the occurrence of interrupts. Node–to–network bandwidth, typically limited by the I/O bus, is also significant for a few applications, but interrupt cost is important for all the applications we study. These results suggest that system designers should focus on reducing interrupt costs to support SVM well, and SVM protocol designers should try to avoid interrupts as possible, perhaps by using polling or by using a programmable communication assist to run part of the protocol avoiding the need to interrupt the main processor.

1.4 Network Interface Support for SVM

In Chapter 5, we focus on improving performance by enhancing the *functionality* of the system layers, driven by the availability of novel, general–purpose mechanisms in commodity network interfaces. Since we use a real system as our vehicle, the performance parameters of the different layers are held fixed.

Since the beginning of SVM [59], much excellent research has been done in improving the functionality of the protocol layer either by itself, based on increasingly relaxed memory consistency models [5, 51], or to take advantage of specialized features of a network interface [44, 56, 7]. Recently, several developments have also occurred in communication layers. Commodity network interfaces that support, or can be programmed to support, key data movement and synchronization mechanisms that do not interrupt the processor have been developed [37, 17]; communication libraries [27, 69, 23, 93, 4, 71] and an industry standard applications programmer interface (API) [26] that incorporate these mechanisms have also been developed.

The focus of Chapter 5 is to understand the extent to which general–purpose mechanisms can be used in commodity network interfaces, in conjunction with the appropriate protocol enhancements, to make large improvements in the performance of SVM systems on clusters. The specific network interface mechanisms that we examine include support for remote write and remote fetch operations without interrupting the remote processor, and for mutual exclusion (locking) in the network interface. Protocol operations like performing diffs and keeping track of versions and coherence information are not performed in the network interface but rather on the main processor; the network interface is assumed to be (and, in our case, is) slow at these activities. The focus is only on general purpose data movement and synchronization mechanisms. These mechanisms do not require a programmable processor on the network interface; we use a programmable network interface only to prototype and evaluate the mechanisms on a real platform. Having shown in Chapter 4 [12] that interrupts and scheduling can be a major problem for SVM, and many efforts have been made to deal with protocol handler scheduling on SMP nodes [31], in this chapter we propose a protocol that eliminates the need for asynchronous protocol processing. Each of our mechanisms reduces the need for interrupts and asynchronous protocol handling; the final protocol (SVM-NI) does not use interrupts for protocol processing at all, and in fact eliminates the need for asynchronous protocol processing.

We demonstrate the benefits of the proposed network interface and protocol extensions through real implementation on a cluster of Intel Pentium Pro SMPs connected together by a Myrinet network [17]. The SVM system that we start from is the HLRC-SMP as presented in Chapter 3. The HLRC-SMP system uses the network interface only as a pathway for traditional send/receive communication. It does not rely on any specific network interface features in its design, and is applicable to any modern system area network. Communication and protocol processing is managed by the same processors that perform computation.

Starting from this system, we use the Myrinet network interface [17], to implement general data transfer and synchronization mechanisms one by one and examine their cumulative impact on end performance; some of these mechanisms are available in the Virtual Interface Architecture (VIA) industry standard API [26]. We leverage the VMMC communication library [23] that provides the software communication architecture needed to exploit some of the features. Also, since taking advantage of some of the mechanisms either requires or is helped greatly by changes to the protocol layer as well, with each mechanism we discuss and implement the relevant protocol changes. Performance is achieved through the synergy of the two system layers, though driven by the communication layer. We do not modify the application layer, relying on the application restructuring for SVM that was performed in [47].

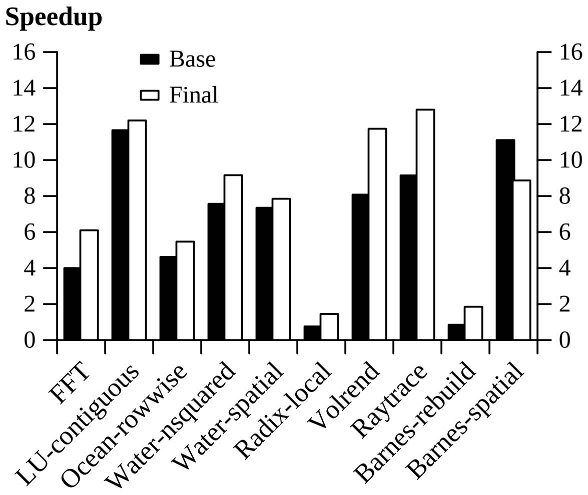

Figure 1.3: Application speedups. The left bar is the base protocol and the right bar the protocol with the NI extensions.

We find that implementing these mechanisms in the network interface, together with the relevant protocol changes, improves performance substantially on the cluster of SMPs for our suite of ten applications. Several applications that originally performed poorly now perform much better, even well, on a 16–processor system, with most applications now yielding speedups close or more than 10 on 16 processors (see Figure 1.3). Application performance improves by (min -20.16%, avg 37.68%, max 117.6%) [1] across all applications. Similarly, the specific components of execution time targeted by the individual mechanisms improve substantially: data wait time improves by (min 2.5%, avg 29.2%, max 45.3%) and lock time by (min 1.9%, avg 35.01%, max 62.7%).

The results indicate that providing support for these features in a network interface is indeed important for SVM performance and for improving clusters as a computational infrastructure. Among the mechanisms we implement, different ones

[1] Certain statistics are reported as a triple of (minimum, average, maximum) values over a range, instead of just reporting only one of these.

are important for different applications, indicating that they should all be supported and leveraged. While speedups are improved significantly by these techniques, they are still not quite as good as those on hardware–coherent systems, even at this 16–processor scale. We use a firmware performance monitor [60], that is integrated with the communication layer, to understand the drawbacks of the system and identify some protocol tradeoffs that bear further investigation. Most importantly, we use the monitor to investigate the causes of the key remaining overheads for SVM performance after all these enhancements, and hence the areas that should be addressed for the next major improvements in bringing SVM performance closer to that of hardware DSM machines.

1.5 Discussion and Conclusions

Chapter 6 discusses a few issues about the simulator and how it validates against the actual system implementation, gives some future work in related research areas and presents the high level conclusions of this dissertation.

Chapter 2

Efficient and Reliable Communication

Low cost and high performance are the potential advantages that motivate building high–performance servers or parallel computing platforms out of a network of commodity computer systems. Such architectures can cost substantially less than custom–designed ones because they can leverage high–volume commodity hardware and software. They can also deliver high performance because commodity hardware and software track technologies well.

Research efforts to provide fast, loosely coupled multicomputer interconnection networks for clusters of workstations at low cost have lead to the design of network interfaces that support minimal standard functionality in the network interface and use powerful basic blocks that allow communication system designers to implement the functionality needed by different paradigms and systems using these basic blocks. These network interfaces usually encompass a general purpose processor, use simple switches to route packets to their destination, and allow for arbitrary network topologies. These characteristics result in the absence of reliability in the network interface hardware and the need to support routing for arbitrary, dynamic topologies in the communication system.

A key enabling technology for this approach is a high–performance communication mechanism that supports protected, user–level message passing. In order for the communication mechanism to be non–invasive with respect to the commodity hardware and commodity operating system software, it must require no modifications to these components. This implies that the communication mechanism should support protection among multiple processes and should be able to deal with communication among virtual address spaces without using special scheduling policies. In order to achieve low latency and high bandwidth, the communication mechanism should also reduce message–passing software overheads to a minimum. This implies support for user–level message passing.

Moreover, a challenging issue in the design and implementation of user–level communication systems is where and how to incorporate a retransmission protocol in order to provide low–overhead, reliable communication for system area networks. Traditionally, reliable end–to–end connections have been implemented in the TCP layer with retransmission. This is a satisfactory design for wide area networks but the overhead of TCP is significant, in particular, when it is implemented on the host computer. This is because a retransmission protocol requires a sender process not to touch its send buffer until the message is acknowledged. The sender has to either wait for the acknowledgment, copy the send buffer to a system buffer, or copy–on–write for the send buffer. Our approach is to implement the standard retransmission method at the data link level, that is, between network interfaces. This approach takes advantage of the buffering of outgoing packets on the network interface and eliminates the need for the sender to wait for acknowledgments or to copy send buffers. We show that the loss of bandwidth of this method is small and its impact on latency is only a few microseconds.

Finally, the nature of these new programmable network interfaces is such that the burden of determining the routes to each host in the network and actually routing the packets to their destinations is left to the communication libraries. Most SAN networks use switches and source routing to connect multiple nodes in a network and to deliver packets to their destinations. We present a new method for determining the topology of the network and the routes to each node in the system. Our method determines the network topology dynamically on demand and supports arbitrary changes in the network configuration.

The goals of this chapter are threefold. It demonstrates that (i) low–latency, high–bandwidth communication can be provided with off–the–shelf components, (ii) reliable communication can be provided at minimal cost, (iii) support for dynamic system configuration can be provided practically at no cost. We discuss how these goals can be achieved and we demonstrate the benefits of these approaches by implementing VMMC [25] on a Myrinet network; using other existing or new communication layers is also possible. The implementation of VMMC on Myrinet achieves about 11 μs one–way latency and provides about 100 MBytes/s user–to–user bandwidth, whereas the cost of reliable communication and support for dynamic system configuration is minimal; latency is increased by less than 2.5 μs, and bandwidth is reduced by less than 8%.

2.1 Myrinet

Myrinet is a high–speed local or system area network for computer systems. A Myrinet network is composed of point–to–point links that connect hosts and switches. The network link can deliver 1.28 Gbits/sec bandwidth in each direction [17].

The Myrinet network provides in–order network delivery with low bit error rates. On sending, the 8–bit CRC is computed by hardware and is appended to the packet. On a packet arrival, CRC hardware computes the CRC of the incoming packet and compares it with the received CRC. If they do not match, a CRC error is reported.

We designed and implemented the VMMC mechanism for the Myrinet PCI network interface. The PCI Myrinet network interface is composed of a 32–bit control processor called LANai (version 4.1) with 1 MByte of Static Random Access Memory (SRAM). The SRAM serves as the network buffer memory and also as the code and data memory of the LANai processor. There are three DMA engines on the network interface: two for data transfers between the network and SRAM and one for moving data between the SRAM and the host main memory over the PCI bus. The LANai processor is clocked at 33 MHz and executes a LANai control program (LCP) which supervises the operation of the DMA engines and implements a low–level communication protocol. The LANai processor cannot access the host memory directly; instead it must use the host–to–LANai DMA engine to read and write the host memory. The internal bus clock runs at twice the CPU clock letting the two DMA engines operate concurrently. The Myrinet network provides network bandwidth of about 160 MBytes/s. However, the total user–to–user bandwidth is limited by the PCI–to–SRAM DMA bandwidth.

Figure 2.1 presents the bandwidth achieved by the host–to–LANai DMA engine for various block sizes. The maximum bandwidth of the PCI bus is close to the 128 MBytes/s which is achieved for 64 KBytes transfer units. However, without hardware support for physical DMA scatter–gather, any communication library which supports virtual memory and direct sending from arbitrary user–level data structures cannot use transfer units larger than the size of a page (4 KBytes). This is because consecutive pages in virtual memory are usually not consecutive in the physical address space.

As a result, user–to–user bandwidth is limited by the 110 Mbytes/s host–to–LANai bandwidth that can be achieved with a 4 KBytes transfer unit.

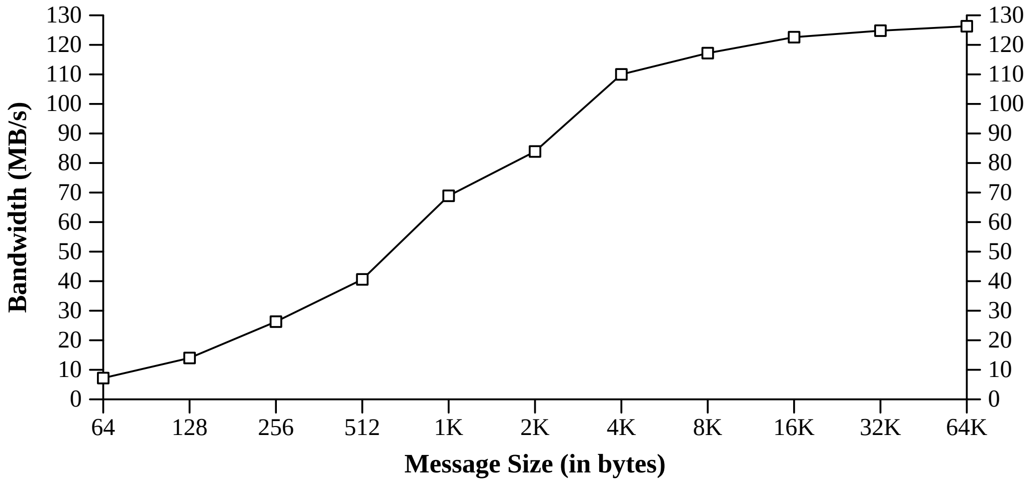

Figure 2.1: Bandwidth of DMA between the Host and the LANai.

The estimate of minimal latency imposed by the hardware includes the time to post a send request, the time needed for sending a one–word message between two LANais and the time to DMA data into the receiver host memory. We measured the costs of memory–mapped I/O over the PCI bus; a read costs 0.422 μs, and a write 0.121 μs. Posting a send request costs at least 0.5 μs, if only writes are used (assuming reads are removed from critical path by preallocating items on send queue). Picking up a send request, preparing a network packet, initializing the DMA network transfer and receiving data on the receiving LANai cost about 2.5 μs. Another 2 μs are needed on the receive side to arbitrate for the I/O bus, initiate the receive host DMA and put data in the host memory. Adding all these overheads together, results in a minimum of about 5 μs.

2.2 Virtual Memory Mapped Communication

Virtual memory–mapped communication (VMMC) [25] is a communication model that provides direct data transfers between the sender's and receiver's virtual address spaces. This section provides a high–level overview of VMMC as implemented on Myrinet hardware. The model has been designed and implemented for the SHRIMP multicomputer [16, 14, 32, 25]. Since the SHRIMP network interface supports VMMC mostly in hardware, the implementation requires either no software overhead or only a few user–level instructions to transfer data between the separate virtual address spaces of two machines on a network. In short, VMMC on the customized network interface of SHRIMP has somewhat better performance, at the cost of more operating system modifications and substantially reduced flexibility.

VMMC provides support for protected, user–level message passing [25, 32]. The main idea is to allow data to be transmitted directly from a source virtual memory to a destination virtual memory. For messages that pass data without passing control, the VMMC approach can completely eliminate software overheads associated with message reception. The VMMC model eliminates operating system involvement in communication while at the same time providing full protection. It supports user–level buffer management and zero–copy protocols. Most importantly it minimizes software overhead. As a result, available user–to–user bandwidth and latency are close to the limits imposed by the underlying hardware.

The VMMC mechanism achieves protection by requiring that data transfer may take place only after the receiver gives the sender permission to transfer data to a given area of the receiver's address space. The receiving process expresses this permission by `exporting` areas of its address space as receive buffers where it is willing to accept incoming data. A sending process must `import` remote buffers that it will use as destinations for transferred data. An exporter can restrict possible importers of a buffer; VMMC enforces the restrictions when an import is attempted. After a successful import, the sender can transfer data from its virtual memory into the imported receive buffer. VMMC makes sure that the transferred data does not overwrite any memory locations outside the destination receive buffer.

Imported receive buffers are mapped into a `destination proxy space` which is a logically separate special address space in each sender. It can be implemented as a subset of the sender's virtual address space or as a separate space (this is the case with the Myrinet implementation of VMMC for SHRIMP). Destination proxy space is not backed by local memory and its addresses do not refer to code or local data. Instead, they are used only to specify destinations for data transfer. A valid destination proxy space address can be translated by VMMC into a destination machine, process, and memory address.

As implemented on Myrinet, VMMC supports the deliberate update mode of data transfer. Deliberate update is an explicit request to transfer data from the sender's local virtual memory to a previously imported receive buffer. The basic deliberate update operation is as follows:

$$\texttt{SendMsg}(\texttt{srcAddr}, \texttt{destAddr}, \texttt{nbytes})$$

This is a request to transfer *nbytes* of data from the sender's virtual address *srcAddr* to a receive buffer designated by *destAddr* in the sender's destination proxy space. When a message arrives at its destination, its data is transferred directly into the memory of the receiving process, without interrupting the receiver's CPU. Thus, *there is no explicit receive operation in* VMMC. VMMC also provides `notifications` to support transfer of control and to notify receiving processes about external events.

Attaching a notification to a message causes the invocation of a user–level handler function in the receiving process after the message has been delivered into the receiver's memory.

2.2.1 Implementation

Our implementation of VMMC on Myrinet consists of a number of software components cooperating with each other. Trusted system software includes (i) the kernel–loadable VMMC device driver, and (ii) the VMMC LANai control program (LCP) which actually implements VMMC. The user–level software consists of the VMMC basic library. A user program must link with it in order to communicate using VMMC calls.

In brief, user programs submit export and import requests to the driver. The drivers and the LANai control programs communicate with each other to match export and import requests and establish export–import relation by setting up data structures in the LANai control program. After a successful import, the user process submits send requests directly to the LANai processor by writing to an entry in the process send queue which is allocated in the LANai SRAM. A request consists of a local virtual address specifying the send buffer, the number of bytes to send and the destination proxy address which is used by the LANai to determine the destination node and physical address. The VMMC LANai control program picks up a send request, translates the virtual send buffer address into a physical address and then sends the data to the destination node. The Myrinet driver assists the LANai control program in the virtual to physical translation of send addresses, if such assistance is necessary, i.e. the translation entry is not present in the LANai SRAM software TLB. The Myrinet driver also implements notification delivery to user processes.

Establishing export–import relations: VMMC maintains a set of page tables in each network interface. The incoming page table (one per interface) has one entry for each physical memory frame, which indicates whether a given frame can be written by an incoming message and whether a notification should be raised when a message arrives.

There is also an outgoing page table, which is allocated separately for each process using the VMMC on a given node. An entry in this table corresponds to a proxy of a receive buffer imported by this process. The format of this entry is a 32–bit integer which encodes the destination node index, the receive buffer virtual address, and the size of the receive buffer.

The size of the incoming and outgoing page tables limits the total size of imported receive buffers. The current limit is 256 MBytes.

The total size of memory that can be exported/imported depends mostly on the amount of SRAM that is devoted to tables and less on the division of the 32 proxy address between the number of buffers that can be exported/imported and the size of each buffer. With 256 KBytes of SRAM devoted to translation tables, 256 MBytes of memory can be exported/imported in the system. Another restriction is that export buffers must be pinned and, Linux allows for only half the physical memory to be pinned.

To export a receive buffer, a user program contacts the local VMMC driver which locks the receive buffer pages in main memory and sets up entries corresponding to the receive buffer pages in the incoming table to allow data reception. On an import request, the importing node LCP obtains the physical addresses of receive buffer pages from the driver on the exporting node. Next, the importing node sets up outgoing page table entries for the importing process that point to receive buffer pages on the remote node.

When sending, the user process passes a proxy address that specifies the destination for the data. A proxy address consists of a proxy page number and an offset within the page. We ensure that a process can send only to valid destinations. Since the outgoing page table is local to the sending process, there is no way a process can use outgoing page table entries set up for others.

Sending and receiving data: Each process has a separate send queue allocated in the network interface SRAM. To submit a send request, the user process writes the next entry in the send queue with the length of the data to be sent and a proxy address which specifies the destination, as described before.

There are two types of send requests: short and long. The short request sends a small amounts of data (currently up to 128 bytes) by copying it directly into the send queue in the LANai memory (no host–LANai DMA is used in this case). With the long send request (up to 4 MBytes), the user process passes only the virtual address of the send buffer. The existence of two send formats is transparent to user programs. The VMMC basic library decides which format to be used for a particular *SendMsg* call.

Messages longer than 4 KBytes are sent in chunks. Each chunk consists of routing information, a header, and data. The routing information is in standard Myrinet format. The header includes the message length and two physical destination addresses. The two addresses are needed in order to perform two piece scatter in the case when the destination memory spans a page boundary. When no page boundary is crossed, the second physical address is set to zero. The sending node LANai prepares each header using the send proxy address passed by the user with the request. With this address, the LANai indexes the outgoing page table to determine the destination node and physical addresses. This information must have been set up on import, as described above, for the proxy address to be valid.

The LANai processor maintains in SRAM virtual–to–physical two–way set associative software TLB for each process using the VMMC on a given node. This TLB is rather large—it can keep translations for up to 32 MBytes of address space assuming 4 KByte pages. The LANai processor uses it on long sends to obtain the physical address of the next page to be sent. If the translation is missing, the LANai raises an interrupt and the VMMC driver provides the necessary translation to update the SRAM TLB. On one interrupt, translations for up to 32 pages are inserted into the SRAM TLB. Send pages are locked in memory by the VMMC driver when it provides the translations for the SRAM TLB. For an improved version of the send TLB mechanism, the reader is referred to [23].

After obtaining the physical address of the next source page, the LANai sends the next chunk of a long message. The chunk size is currently set to the page size (4K bytes), except for the first chunk which is equal to the amount of data needed to reach the first page boundary on the send side. The sending of each chunk of a long message requires two DMA transactions: host to LANai buffer and LANai buffer to network. These transactions are pipelined to improve bandwidth. Additionally, since the network bandwidth is higher than the PCI bus bandwidth, we prepare the header for the next chunk when the network DMA of the previous chunk has finished, but while the host DMA of the current chunk may still be in progress.

When the last chunk of a long message is safely stored in the LANai buffer, the LANai reports (using LANai to host DMA) a one–word completion status back to user space. This technique allows the user program to spin on a cache location while waiting for message completion, and it avoids consuming memory bus cycles.

On arrival of a message chunk, its data is scattered according to the two physical addresses included in the header. The LANai is able to compute scatter lengths using the total message length and the scatter addresses.

2.2.2 Performance

Our implementation and experimentation environment consists of 16 PCI PCs connected to Myrinet switches via Myrinet PCI network interfaces. In addition, the PCs are also connected by an Ethernet. Each PC is a Dell Dimension P166 with a 166 MHz Pentium CPU with 512 KByte L2 cache. The chipset on the motherboard is the Intel 430FX (Triton) chipset. Each PC has 64 MBytes of Extended Data Out (EDO) main memory and 2 GBytes of EIDE disk storage.

Each node in the cluster runs a slightly modified version of the Linux OS version 2.0.24. The modifications are minimal and are needed to extend the device to kernel interface by exporting a few more symbols for loadable modules. Linux provided us with most of the functionality we needed, including calls to lock and unlock pages in physical memory. Loadable kernel modules proved to be a useful and powerful feature of Linux.

The new kernel–level code we needed is implemented in a loadable device driver including a function which translates virtual to physical addresses and code that invokes notifications using signals. We note here that we could have completely avoided any modification of the OS if we had provided a special variant of *malloc* which would allocate send and receive buffers in driver–preallocated memory mapped into user space. This solution, however, has the serious drawback of not supporting direct send and receive from user static data structures (which are not dynamically allocated).

Micro–benchmarks

The basic operation in VMMC is *SendMsg*. In an effort to capture the different cases that arise in message passing programs we analyze different forms and aspects of this basic operation. When a virtual–to–physical translation is needed on a send, we make sure that it is present in the LANai software TLB. This eliminates the cost of interrupting the host to ask its kernel for the missing translations. We feel this assumption is reasonable, as the LANai TLB can keep mappings for 32 MBytes of user space.

We first present the one–way latency and its cost breakdown for small messages. Then we present the overhead of the send operation in two cases: *synchronous sends,* where a send operation returns only after the data is transferred to the network interface and the send buffer can be safely reused, or *asynchronous sends,* where a send operation returns immediately and if the sender needs to reuse the send buffer, it first has to verify that it is safe to do so. Finally we present three different types of bandwidth that try to capture different types of traffic in the network: *Unidirectional* is the bandwidth of a one way transfer. Data is flowing in one direction only and the sender does not need to wait for the receiver to acknowledge a message before it sends the next one. In the *Bidirectional ping–pong* benchmark data flows in both directions in a ping–pong fashion. After sending a message, each sender waits until it receives a message before sending again. One of the nodes starts as a sender and the other as a receiver. *Bidirectional simultaneous* is similar to the bidirectional ping–pong benchmark, only that both nodes start as senders. This benchmark can potentially use all three DMA engines of the network interface, depending on how the Myrinet firmware is written.

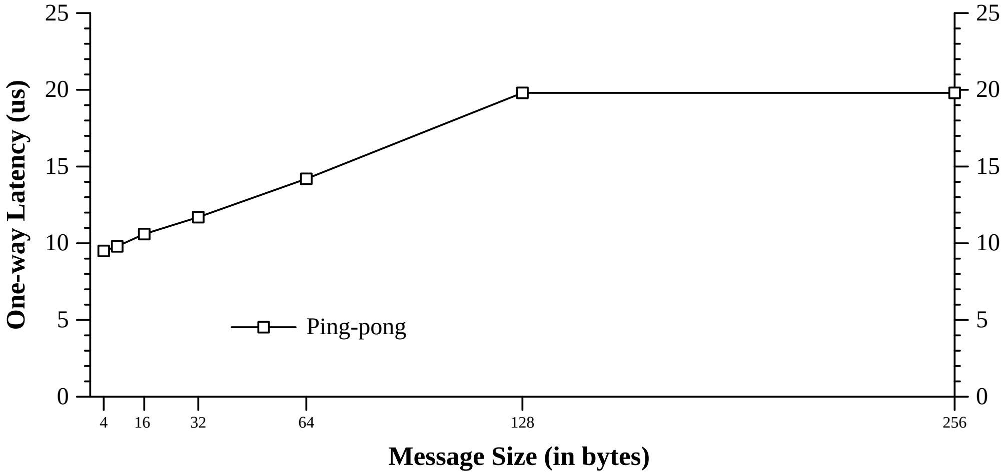

Figure 2.2: VMMC latency for short messages.

Latency: Figure 2.2 gives one–way latency for small messages. One–word latency is about 11 μsec. Messages of up to 32 words are copied to the SRAM send queue using memory–mapped I/O and then the LANai copies message data into the network buffer. For longer messages, host–to–LANai DMA copies data from a host send buffer to a network buffer.

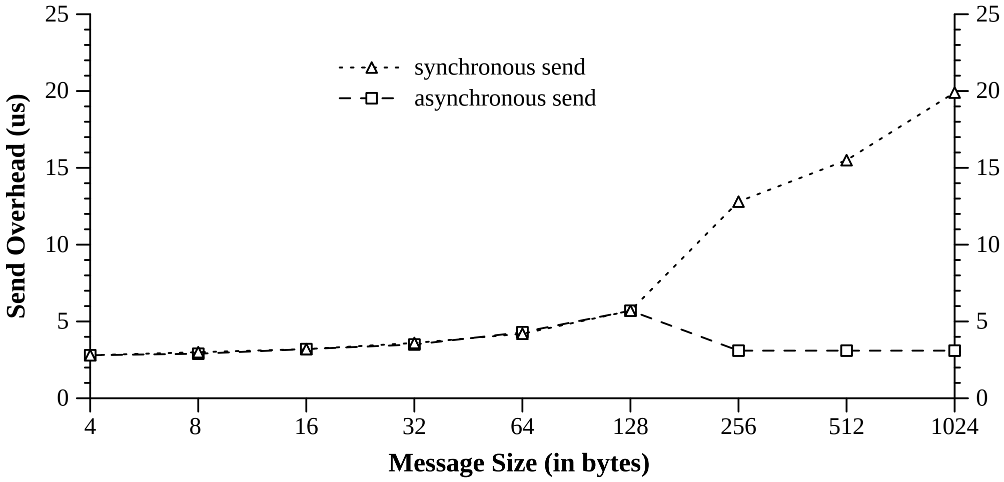

Figure 2.3: Overhead of the synchronous and asynchronous send operations.

Synchronous send overhead: An important aspect of the synchronous send operation is the cost of copying the data to the network interface. In this case we measure the time it takes to transfer the data to the network interface, at which point it is safe to reuse the send buffer. Figure 2.3 shows that synchronous send overhead is about 3 μsec and grows slowly up to 128 bytes, but for longer messages there is a significant jump in overhead caused by the change of protocols and the use of host–to–LANai DMA on the send side. Synchronous send overhead, not latency is the motivation why the threshold at which send protocol changes from short send to long send is not lower than 128 bytes. Setting this threshold to 64 would increase synchronous send overhead for messages between 64 and 128 bytes long, although latency would not change much, as shown on Figure 2.2. On the other hand, we cannot set this threshold higher than 128 bytes because of limited size of LANai SRAM.

Asynchronous send overhead: In many applications, it is useful to be able to post a send request and then proceed with computation. In this case the send buffer cannot be reused, unless the application verifies that it is safe to do so by using another call to the network interface to check the status of the posted request. Figure 2.3 shows that the asynchronous overhead of a long send is slightly lower than for short sends. This is because the long send request is of fixed size and does not require data copying, whereas on a short send the host has to copy data with memory–mapped I/O to the LANai buffer. We note, however, that asynchronous and synchronous send overheads for short sends are equal, since in both cases the host executes the same code while the synchronous overhead of a long send is much higher than its asynchronous overhead.

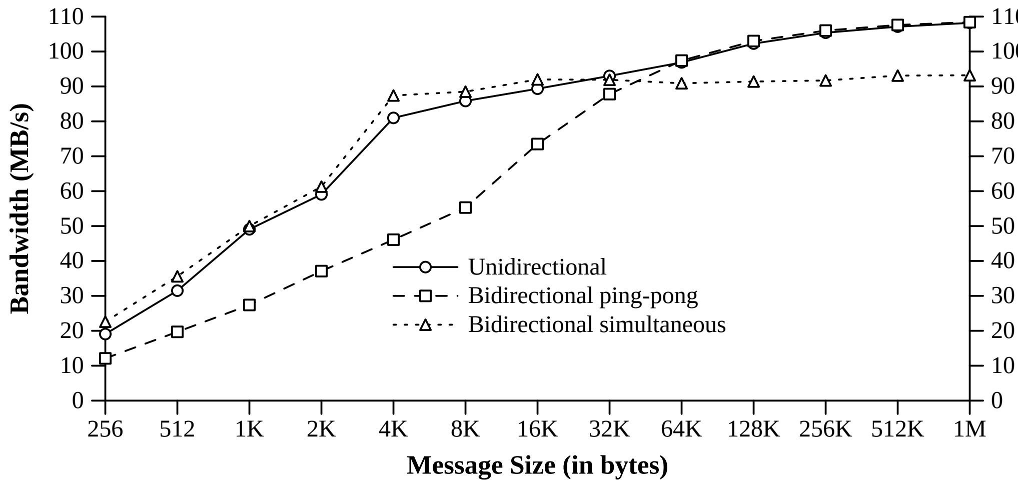

Figure 2.4: VMMC bandwidth for different message sizes.

Unidirectional bandwidth: This benchmark shows (Figure 2.4) how fast one node can push data to the network. We see that this type of bandwidth raises quickly and reaches a maximum of about 110 MBytes/s, since there is no synchronization involved and the sender does not need to wait for the receiver. Moreover there is no contention in the network interface, as only one node is sending data.

Bidirectional ping–pong bandwidth: This is the traditional ping–pong benchmark. Figure 2.4 shows the bandwidth for varying message sizes. The highest bandwidth provided by VMMC is about 100 MBytes/s. As explained, the available bandwidth is limited to 110 MBytes/s, with our implementation delivering 120 MBytes/s of this limit. Such good performance results from 1) a tight sending loop, 2) pipelining the host send DMA with the net send DMA and 3) precomputing the headers as explained above. However, this loop also needs to be responsive to unexpected, external events, such as the arrival of incoming data packets. In this event, we abandon the tight sending loop and return to the main loop of the VMMC LANai control program.

Bidirectional simultaneous bandwidth: This benchmark tries to capture the behavior of VMMC in the presence of simultaneous bidirectional traffic. Both sides start sending at the same time, they wait until they have received the message from the peer process and then go on to the next iteration. Figure 2.4 presents the bandwidth in this case. Note that this is the total bandwidth of both senders. The maximum available bandwidth, equal to 91 MBytes/s, is lower than in the previous case. With bidirectional traffic we cannot use the tight sending loop because packets leave and arrive simultaneously. Instead we have to go through the main loop of our software state machine which slightly increases the software overhead, and reduces the bandwidth.

2.3 Reliable Communication

In communication systems, the fault model refers to the level of reliability provided. Previous work has shown that the fault model of a communication system is an important issue that requires attention. A strong fault model simplifies substantially application design and results in a usable system. Systems with weak fault models can be very difficult to program, even for relatively simple applications. Ignoring error handling in the network layers and leaving it to the applications requires substantially more effort on the programmer's side. In general it has been argued that the communication subsystem should provide reliable communication for most cases.

There are two approaches to reliable communication: (i) High performance systems provide reliable communication at the lowest possible level, in hardware. In these systems the network interfaces are responsible for providing the user with reliable communication channels. Communication systems that are built on top of these interfaces treat network errors as catastrophic. This approach works well as long as network errors are extremely rare or when the packet retransmission is handled by the hardware (like S–Connect [66]). However, providing reliability in the physical layer leads to expensive hardware and limits the design choices significantly. (ii) In local and wide area networks, reliable communication is implemented in software in some layer of the communication stack (e.g., TCP) Each approach is targeted for a set of specific needs. In local and wide area networks the ability to tolerate higher performance penalties and the need for non–uniform commodity hardware and cost limitations allow and demand solutions at higher software layers. In high performance networks, the same factors lead to very efficient, custom and highly tuned solutions in hardware.

With the emergence of high performance commodity workstations and system area networks it is possible to provide supercomputer performance on clusters of high–end systems, composed completely of commodity parts. A key issue towards that direction has been to provide the users with very efficient, user level communication libraries. In these libraries, support for reliable communication does not fall in any of the two previous categories. Although error rates in the network are very small, errors do happen and as a result the hardware does not provide reliable transmission. For Myrinet however, errors were observed relatively often (on a daily basis) and cannot be ignored. These new, user–level communication libraries strive for high performance and integration with a traditional fault tolerance mechanism can compromise performance. Thus, the decision of where to incorporate fault tolerance is critical for system complexity and performance.

Most high–level communication APIs provide reliable communication to the user by implementing retransmission in one of the software layers of the protocol stack. Stream sockets, for example, are typically built on top of TCP/IP with reliable end–to–end connections implemented in the TCP layer. This design is suitable for wide area networks since network connections go through intermediate host systems and packets may need to be stored for a long time before they are acknowledged. However, this results in high overhead for TCP; highly tuned implementations add over a hundred microseconds of overhead.

A system area network (SAN) is different from a local or wide area network. A SAN typically has a small diameter (a few feet) and is used to connect computer systems to form a high–performance server. Applications on such networks of computers require low–latency and high–bandwidth communication. A key issue that arises is where and how to provide reliable communication in the system so that high–level APIs can be implemented without compromising performance.

In a system area network, reliable communication can be provided at different levels. It can be incorporated in either the user–level library that runs on the host, or in the part of the library that runs in the network interface as firmware. We ruled out the design choice of implementing reliable communication at the user–level library. The main reason is that a retransmission protocol requires the sender process not to touch its send buffer until the acknowledgment is received. The sender has either to wait for the acknowledgment, copy the send buffer to a system buffer, or use copy–on–write for the send buffer. The overhead of these design choices is significantly high.

In this work we argue for providing support for reliable communication in the network interface code of the communication library. Implementing retransmission at this level takes advantage of the existing buffering scheme for outgoing packets on the network interface and eliminates the need for the sender to wait for acknowledgments or to copy send buffers. Another reason to provide reliable communication at the data link level is that system area networks are simpler than local or wide area networks. Wide area networks operate in a store–and–forward fashion, whereas in a system area network, a packet is typically routed through switches from a single sender to a single receiver without being stored and forwarded by intermediate nodes.

2.3.1 Firmware support for reliable communication

Our scheme deals with transient network failures and provides applications with reliable communication at the data link layer. The goal is to tolerate CRC errors, corrupted packets, and all errors related to the network fabric; links and switches can be replaced on the fly.

Fault model

We distinguish failures in two orthogonal ways. First, a failure can be either a *network failure* or an *end failure.* Network failures are related to the network interconnect. This includes all links and switches that are used in the network. End failures are the ones that are related to process and node failures. Second, an error can be either transient or permanent. A failure is transient, if the failing component becomes operational after a short amount of time. Otherwise, it is permanent. All failures are considered transient first and, if they persist, they are categorized as permanent. VMMC deals only with transient network failures.

The fault model adopted in VMMC is constrained both in terms of design and implementation by many factors. Fault tolerance should not be achieved at the cost of reduced performance. There is limited buffer space on the Myrinet network interface (currently 1 MByte) and the processor on the network interface is a relatively slow processor. A complicated protocol would increase the occupancy of the processor and impact performance. The network we assume is a LAN within a building or other facilities of comparable size. The average meantime between failures is rather big (in the order of days).

As mentioned VMMC deals only with transient network errors. If transient errors become permanent, the remote node is declared unreachable, the state that is related to the remote node is cleared and the user is notified at the next send (or other) operation. Packets that cannot be delivered to the remote node are dropped. In this case the user needs to reestablish the mappings in order to resume communicating with the remote node. This means that the user does not know exactly which packets were delivered and which not. However, if a packet is dropped, all subsequent packets will be dropped as well. The semantics are the same as in TCP/IP.

Implementation

VMMC implements a simple retransmission protocol at the data link layer–between network interfaces. Our scheme buffers packets on the sender side and retransmits when necessary. Each node maintains a retransmission queue for every node in the system. The available buffer space is managed dynamically so that no buffers need to be reserved for each node. Each packet carries a unique (for the sender–receiver pair) sequence number. Sequence numbers and retransmission information is maintained on a per–node and not per–connection basis. The receiver acknowledges packets. Each acknowledgment received by the sender, acknowledges (and frees) all previous packets up to that sequence number. There is no buffering at the receiver. If a packet is lost, all subsequent packets will be dropped. However, if a previously acknowledged packet is received again, it is acknowledged. In the current implementation, there are no negative acknowledgments for lost packets.

The retransmission scheme will always deliver packets in the presence of transient network failures. If there are no buffers on the sending side, the senders will back off and wait until space is available on the network interface. The receive side can always receive or drop packets, since the sender will retransmit if packets are not acknowledged.

The number of buffers available at the send side affects system performance. As mentioned above our network interface cards have 1 MByte of SRAM, which provides more than 100, 4 KByte–buffers for buffering in the retransmission scheme. These buffers are used dynamically so that no buffers need be reserved for each node.

Although the retransmission scheme will never fail, we should note that malicious processes can temporarily (for a few seconds) hurt the performance of the system by needlessly occupying buffers until the system recovers the buffers with a timeout mechanism. This is not a matter of correctness, but rather fairness in the system.

Acknowledgments can be sent either explicitly or piggy–backed to messages that are transferred in the opposite direction. Piggy–backing reduces traffic and overhead at the LANai significantly, but is useful only when messages are sent in both directions. One way traffic always needs explicit acknowledgments. Explicit acknowledgment messages do not carry sequence numbers and they are not acknowledged. If a packet is lost and the sender does not receive an acknowledgment for it, it will timeout and retransmit the lost and all subsequent packets.

We use two methods for deciding when explicit acknowledgments need to be sent. The first one, specifies that an explicit acknowledgment will be sent every n unacknowledged packets. The selection of n is very important to system performance and scalability. In the case of one way traffic, a large n would require a lot of buffering on the sending side. On the other hand, a small value of n generates a lot of extra traffic; more importantly it increases the occupancy of the network interface which must process the frequent acknowledgments.

The second method requires that the sender specify when the receiver needs to send an acknowledgment. If there have been no piggy–backed acknowledgments, and the available buffer space in the sender is reduced below a threshold, then the sender notifies the receiver with a bit in the packet header, that the receiver needs to acknowledge the packets it received, so that buffer space can be recovered in the sender. We implement and present results for both methods.

Ordering of packets is maintained with 16 bit sequence numbers. The system deals with wrap around in sequence numbers. We define the boolean function *more recent* $MR(x, y)$ as:

$$MR(x, y) = ((x > y) * (x < y + c)) + ((x < y) * (x < y - c))$$

$MR(x, y)$ specifies when a packet with sequence number x is strictly more recent than a packet with sequence number y. c is a constant that depends on the amount of buffer space available in the system and specifies the biggest possible distance between the sequence numbers of any two packets (between a pair of nodes) that are *alive* in the system at any time. A packet is considered if it has been sent and the corresponding buffer is not yet reclaimed at the sender.

Since the LANai processor is relatively slow, it is desirable to remove costly operations from the critical path. A number of implementation optimizations are used to distribute the costs uniformly in the execution path in the LANai.

2.3.2 Performance

Table 2.1 shows the basic communication performance of VMMC with and without support for fault tolerance [1]. As mentioned previously, the frequency at which ac-

[1]The numbers reported in Table 2.1 for VMMC without reliable communication are for a version of VMMC that includes some more functionality than the basic mechanisms described in the previous section [23]. The reliable communication mechanisms have been implemented on this version of VMMC.

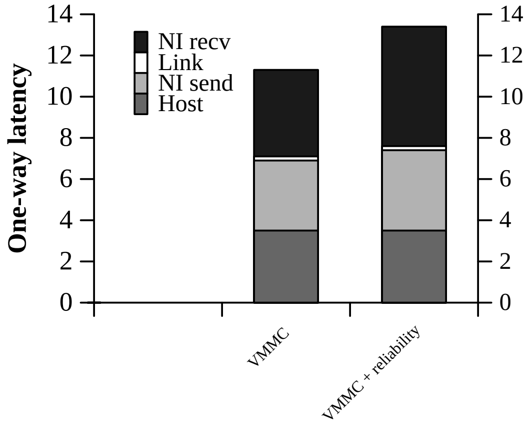

Figure 2.5: Latency breakdown for one–word messages.

knowledgments are sent is an important parameter in the system. The same Table shows also the performance of the system for different acknowledgment frequencies. In the current implementation, we acknowledge every sixth packet (explicitly or with a piggy–backed message) and we use timeouts to handle boundary cases. More frequent acknowledgments reduce the performance of the system because of the extra traffic they generate and the extra processing needed at the sender of each packet. Less frequent acknowledgments generally help, but increase buffer requirements in the system. Piggy–backing reduces the effects of the extra communication, but is useful only in bidirectional communication patterns. Experiments show that it increases bidirectional bandwidth by more than 15%. The second approach we use for determining when explicit acknowledgments are sent, performs the same as our best selection for the fixed threshold method. We see that support for fault tolerance can be provided at low cost. Latency increases by about 2μs (from 11.1μs to 13.4μs) and bandwidth decreases by 2%-10%. The additional overhead comes from moving messages between queues and handling acknowledgments. Figure 2.5 shows the breakdown for one–word messages in VMMC with and without reliability. We see that the overhead of reliable communication is mainly incurred on the receive side.

# packets/ack	NO RC	1	2	4	6	8	10	units
Latency	11.1	23.93	18.29	14.86	13.4	13.4	13.4	μs
Unidir b/w	97.61	82.08	89.97	92.11	92.63	92.81	93.01	MBytes/s
Bidir ping–pong b/w	99.39	80.62	87.37	89.54	90.05	90.26	90.39	MBytes/s
Bidir simultaneous b/w	91.87	80.30	88.42	89.54	90.77	91.05	91.10	MBytes/s

Table 2.1: Basic performance numbers with varying ack window. The default value is one acknowledgment every six packets

2.4 Dynamic System Configuration

Another important part of a communication system is the way routing is performed. Although a lot of work has been done in routing for multiprocessor networks, routing for emerging SAN networks is mostly unexplored. In traditional multicomputer networks with fixed topologies the routing algorithm is usually embedded in the network hardware. Each packet carries a destination address, which is enough to determine the routing path, given the fixed topology of the network.

In networks however, where the topology is not fixed and changes dynamically, other forms of routing are necessary. In these cases, the communication system supports some form of addressing remote nodes. In general, there are two main methods for delivering packets to their destination. First, to have each network element know which link each incoming packets should be sent to. These networks require that intermediate elements are intelligent entities that can take decisions based on routing tables and information in the packet headers. Second, to have each packet specify the full path to the destination node upfront. Whenever the packet reaches an intermediate network element, its header specifies which link this packet should be directed to. This second form of routing is known as *source routing* and uses simple switches to perform routing inside the network.

In system area networks the the routing components are in the network are usually simple and fast switches. For instance, Myrinet supports source routing, with the routes being specified in each packet header by the sender. However, communication systems that are built on top of this type of networks need to be able to determine the routes to any node in the system. This problem has two parts. First, to figure out a graph that represents the topology of the network and second, given a topology, to compute routes. We refer to the first problem as the *mapping problem.* There has been a lot of work in computing routes with specific properties from network topologies [73, 67, 22, 38]. One of the most important properties is that routes should be deadlock free. In this section we present a new method for solving the mapping problem, and we discuss how to solve the problem of deadlock–free routes. In the following paragraphs we discuss some related work in solving the mapping problem for user–level communication systems in SAN networks like Myrinet and then present our solution.

The first solution to the mapping problem is given by Myricom [17]. They use an algorithm that is implemented on the network interface and is integrated with the communication library. This algorithm selects a node as the master node in the system. The master node uses polling to determine the identity of each node in the system and to figure out the topology of the network. The algorithm has to be able to detect cycles and multiple paths to the same node. After the graph that represents the network topology has been determined, it is sent to all nodes in the system. Each node then computes *deadlock–free* routes to all other nodes using the UP*/DOWN* algorithm [67][2]. The Myricom mapping algorithm runs periodically and maps the full network, in the presence of arbitrary traffic in the network.

A second approach is described in [62], where the topology of the network is determined with a different algorithm. Routes are again computed using the UP*/DOWN*

[2]The UP*/DOWN* algorithm divides the network links in two groups (UP and DOWN) and imposes an ordering between these categories. This ordering is used to compute deadlock free routes.

algorithm. Their algorithm also maps the full network, but runs on the host processors (as a normal Active Messages program) and requires that there is no other traffic in the network during the mapping phase.

Most other user–level communication systems that are implemented on top of Myrinet use either static maps or the algorithm provided by Myricom to solve the mapping problem. When static maps are used, the user specifies the topology of the network beforehand. After the communication system is started, no changes in the topology are permitted. Reconfiguring the system (changing the topology, or adding and removing nodes) requires a system shutdown and restart. As the number of nodes in a system increase, this solution is not adequate since it is likely that not all the nodes will be operational all the time. Moreover, maintenance of the system requires all the nodes to be brought down, the system to be partitioned and then part of the system to be brought up again. Clearly, some form of dynamic determination of the network topology is needed.

2.4.1 Firmware support for dynamic system configuration

We take a different approach in solving the mapping problem. Note that in the two previous systems, the map of the full system is constructed each time the algorithm runs. We use an algorithm where routes to each node in the system are determined dynamically, on demand. The basic block in our algorithm is the Myricom approach for polling nodes of the network for their identity. At startup in our system, each node has no information about the topology of the network. Whenever a packet needs to be sent to a node, the network interface firmware starts to poll the nodes in the network, trying to find the destination of the packet. After the destination is found, a route is constructed and stored in a cache, both in the sender and the receiver.

The topology of the network is discovered and the routes are built dynamically, on demand.

This approach has many advantages. Whenever there is a change in the topology of the network, there is no need for the full network to be remapped. If a node is added, it will be discovered the first time a packet is sent to it. If on the other hand a node is removed, when a packet is sent, after the retransmission mechanism fails to deliver the packet the node will be declared unreachable and all its state (including the route) will be removed from the cache in the sender. This allows for nodes to be moved in the network transparently. Also the topology can change in arbitrary ways during system operation. Routes to nodes that change relative position will be invalidated and new ones will be computed. Of course the identity of each node needs to remain the same in the network, regardless of each location.

The algorithm deals correctly with packets from previous node or buffer incarnations in the system, something not at all trivial. The data link layer synchronizes properly the sequence numbers in abnormal cases, without worrying about packet generations. At a higher level in the system, generation numbers are used to deliver or drop packets depending on whether they match with the currently exported buffers. This allows for packets that are delivered to the wrong node because of changes in the topology, to be detected and dropped. Generation numbers and fixed node identities are used for this.

This mapping scheme needs to be (and is) integrated with the retransmission scheme that is used in the firmware. Sequence numbers need to be synchronized whenever there are changes in the topology. Moreover, since routes are computed on the fly, without the full topology of the network being available, deadlocks can occur. Myrinet however, has a timeout mechanism for deadlock recovery (not avoidance or detection). This mechanism is used to break deadlocks when they occur. In our case, we take advantage of this mechanism and the retransmission protocol of VMMC. If a deadlock occurs the receiver will be reset and packets in transit will be dropped. However, these packets will be retransmitted and eventually delivered to the destination.

2.4.2 Performance

Our approach for discovering the network topology and computing routes, does not affect the common path in the communication system and thus does not affect performance when the system topology does not change.

2.5 Final VMMC Performance

In order to understand the performance of VMMC, we measured the communication latency and bandwidth with micro–benchmarks, and analyzed VMMC with the LogP model [20]. Figures 2.6 and 2.7 show the performance of VMMC.

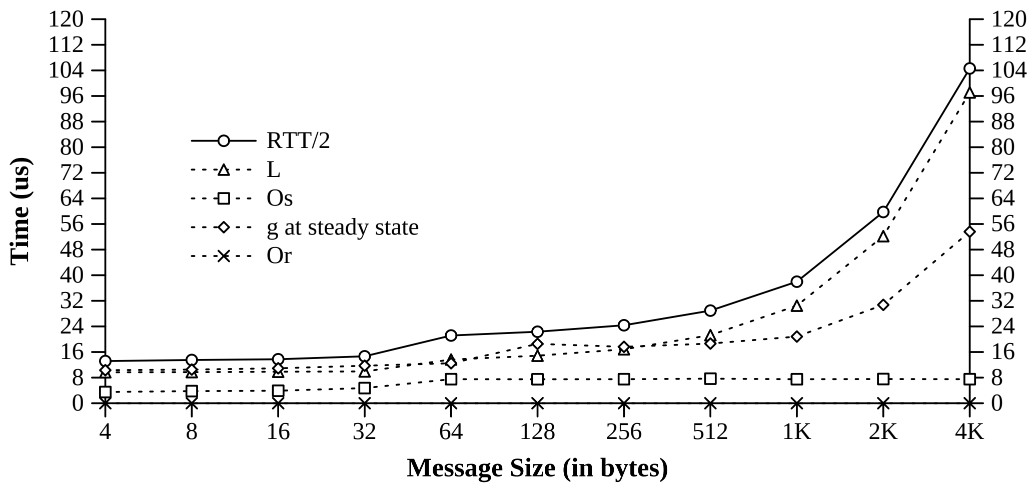

Figure 2.6: LogP numbers

We characterize the performance of our system using the five parameters of the

LogP model: $(\frac{RTT}{2}, L, O_r, O_s, g)$. $\frac{RTT}{2}$ is the one–way host to host latency for a ping–pong test. O_s, O_r are the host overheads of sending and receiving a message respectively. L is the one–way network latency between network interfaces. Finally g (gap) is the time between two successive sends and describes how fast messages can be pipelined through the system.

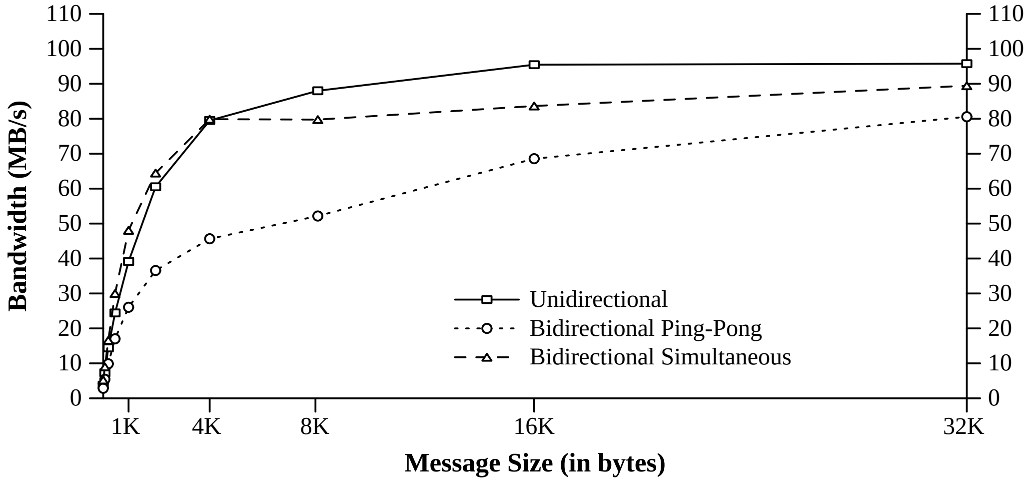

Figure 2.7: VMMC bandwidth numbers.

Besides the LogP characterization we also run the same three bandwidth tests as in Section 2.2. Figure 2.7 shows results for these three types of bandwidth. Bidirectional ping–pong bandwidth grows slower than the other two types because there is only one message in the network at any given time, so pipelining is less effective for short messages. For longer messages, data is transfered in 4 KBytes packets and unidirectional bandwidth catches up with the other two tests. The maximum bandwidth is in the range of 90-100 MBytes/s for all the tests. The 4 KBytes message bandwidth is about 44 MBytes/s for the bidirectional ping–pong test, and about 80 MBytes/s for the other two tests.

2.6 Related Work

The Myrinet switching fabric has been used by several different research projects to produce high performance message passing interfaces. These include the Myrinet API [17] from Myricom, Active Messages [27] (AM) from the Univ. of California at Berkeley, Fast Messages [69] (FM) from Illinois University and PM [93] from the Real World Computing Partnership.

The Myrinet API supports multi–channel communication, message checksums, dynamic network configuration and scatter/gather operations; however, it does not support flow control or reliable message delivery. On our hardware platform the Myrinet API has a latency of 63 μs for a 4 byte packet and a peak ping–pong bandwidth of 30 MBytes/s for an 8 KByte messages.

In Active Messages [20] each communication is formed by a request/reply pair. Request messages include the address of a handler function at the destination node and a fixed size payload that is passed as an argument to the handler. Notification is done using either waiting for response, polling or interrupts. The current implementation of active messages (AM–II) supports multiprogramming, protected communication, reliable message delivery and dynamic mappings. Reliable communication is provided at the library level, by using an extra copy that Active Messages perform for each message that is sent.

Basic Interface for Parallelism (BIP) [71] is a minimal library that aims at providing raw hardware performance to its users. To achieve this, it allows direct access to all system resources and provides data transfer only (no transfer of control). It is intended for single–user systems, and does not support protection, multiprogramming, reliable communication or support for dynamic system configuration. BIP delivers messages in FIFO order and achieves a minimum latency of about 4.5 μs and a maximum bandwidth of about 125 MBytes/s.

Fast Messages (FM) 2.0 [68] is a user–level communication interface which does not provide internode protection, reliable message deliver or dynamic mapping. FM 2.0 supports a streaming interface that allows efficient gather/scatter operations. It also support polling as a notification mechanism, and thread safe execution. FM 2.0 is similar to AM in that each message is associated with a handler that will process the message on the receive side. On our hardware FM has a latency of 10.7 μs for an 8 byte packet and a peak ping–pong bandwidth of 30 MBytes/s for an 8 KByte messages (or a maximum of about 70 MBytes/s if processors that support write combining in the write buffer are used). The low latency is achieved by using a small buffer size (128 bytes) and programmed I/O on the sending side. Using programmed I/O avoids the need for pinning pages on the sender side. On the receiver side, DMA is used to move the message data from the LANai to the receive buffers, which are located in pinned memory. The handlers then copy the data from the receive buffers to the user's data structures. In contrast, VMMC avoids copying on the receiver side by allowing the user to access data directly in the exported (pinned) memory.

PM, like AM and FM, is a user space messaging protocol. It requires gang scheduling to provide protection. PM supports multiple channels, a modified ACK/NACK flow control, and notification by either polling or interrupts. PM runs on hardware similar to ours, where it has a latency of 7.2 μs for an 8 byte message and 118 MBytes/s peak pipelined bandwidth achieved with a transfer unit size of 8 KBytes. PM can use transfer size bigger than a page size because it sends data only from special pre–allocated send buffers. As a result, a user must often copy data on sender side before transmitting it. The cost of this copy is not included in the peak bandwidth number above and it will reduce available user–to–user bandwidth. Even ignoring the cost of copying, when the transfer unit is limited to the page size (4 KBytes) both

PM and VMMC have a pipelined bandwidth close to 110 MBytes/s (see Figure 2.1). PM achieves slightly lower latency than VMMC because it allows the current sender exclusive access to the network interface. This solution involves saving and restoring of channel state on a context switch, an expensive operation. PM does not support reliable communication or dynamic system configuration. Moreover, it is not clear how gang scheduling performs with respect to the communication layer if SMP nodes are used.

A lot of work has been done in providing reliable communication in traditional, kernel–based, communication systems and especially TCP/IP. The Nectar system [1] used a general purpose processor in the network interface to run the TCP/IP protocol stack. However, this is the only work we are aware of, that argues for and incorporates reliable communication in the network interface, in a user–level communication system for SAN networks. AM [63] support reliable communication in the user–level library by performing one copy.

Our work is similar to the work done in the context of Autonet [80, 73, 67]. Autonet was designed as a general purpose LAN to provide an alternative to slow 10 MBit ethernet networks. Autonet supports automatic reconfiguration but does not provide reliable transmission. Unlike Myrinet, Autonet uses smart switches (routers).

2.7 Discussion and Conclusions

This chapter discussed three basic issues in system area network: (i) how can low–latency, high–bandwidth communication be provided with off–the–shelf components, (ii) how can reliable communication be provided at minimal cost, (iii) how can support for dynamic system configuration be provided practically at no cost.

Commodity systems area networks that aim at providing high–performance, usually do not provide the user with reliable communication at the physical layer. This leaves the burden of dealing with errors to the communication library or the application writer. In this work we argue for providing reliable communication in user–level communication systems, in the firmware of the network interface, at the data link layer. This approach takes advantage of the buffering in the network interface to eliminate extra copies. It has a very small impact on performance; about 2μs for latency and less than 10% for all different types of bandwidth. We also present a new solution to the *mapping problem* that takes advantage of the retransmission mechanism in the network interface to simplify certain tasks. This solution, determines the topology of the network dynamically on demand. It allows for changes in the configuration of the system during normal operation. This is very useful for emerging system area networks that are not as tightly coupled as previous multicomputer networks.

The benefits of this approach were demonstrated by presenting the design and implementation of virtual memory–mapped communication (VMMC) [25, 24, 23] on a Myrinet network of PCI–based PCs; using other existing or new communication layers is also possible. VMMC is a communication model providing direct data transfer between the sender's and receiver's virtual address spaces. This model eliminates operating system involvement in communication, provides full protection, supports user–level buffer management, zero–copy protocols, and minimizes the software overheads associated with communication.

The overall communication performance of the VMMC implementation, including the retransmission mechanism and the dynamic mapping features is comparable or better than the performance of systems that do not provide this functionality. Its one–way latency is 13.4 μs for a small messages and it achieves an one–way bandwidth of 80 MBytes/s for 4 KByte messages and over 90 MBytes/s for messages larger or equal to 16 KBytes.

Chapter 3

SVM for networks of SMPs

As the workstation market moves form single processor to small–scale shared memory multiprocessors, it is very attractive to construct larger–scale multiprocessors by connecting symmetric multiprocessors (SMPs) with efficient commodity networks. With hardware–supported cache–coherent shared memory within the SMPs, the question is what programming model to support across SMPs. A coherent shared address space has been found to be attractive for a wide range of applications, and shared virtual memory (SVM) protocols have been developed to provide this model in software at page granularity across uniprocessor nodes. It is therefore attractive to extend SVM protocols to efficiently incorporate SMP nodes, instead of using message passing both within and across SMP nodes [61] or using a hybrid programming model with a shared address space within SMP nodes and explicit message passing across them. The protocols should be optimized to exploit the efficient hardware sharing within an SMP as much as possible, and invoke the less efficient software protocol across nodes as infrequently as possible.

A key question in the communication layer is how much and what kind of hardware support is particularly valuable in improving the performance of such systems. The most popular form of hardware support used so far is the propagation of fine–grained writes to remote memories [16, 50, 37].

This support inspired the design of a family of new, home–based [44, 45, 100, 54] protocols for page–based software shared virtual memory (SVM), which differ from earlier all–software protocols not only in hardware support but also in the manner in which they propagate changes and solve the multiple–writer problem. Essentially, every shared page has a *home* node, and writes observed to a page are propagated to the home at a fine granularity in hardware [44, 46, 54], without interrupting the processor at the home. Shared pages are mapped write–through in the caches so that writes can be snooped off the memory bus. When a node incurs a page fault, the page fault handler retrieves the page from the home where it is guaranteed to be up to date [44, 46]. Data are kept consistent according to a page–based software consistency protocol such as lazy release consistency [52]. Thus, consistency in maintained at page granularity, while there is some hardware support for fine–grained communication. The SHRIMP system [16] provides both the write snooping hardware as well as a customized network interface (NI) to support automatic update. All–software versions of such home–based protocols have also been proposed and implemented [44, 46, 45, 100, 72].

In this chapter we first present the necessary protocol extensions to support this two level hierarchy. We use detailed, application–driven simulations to understand how successful such a protocol might be, and in particular whether and to what extent the use of SMP nodes improves performance over the traditional method of using SVM across uniprocessor nodes. We examine cases where the home–based SVM protocol across nodes is supported entirely in software, and also cases where the propagation of modifications to the home is supported at fine grain in hardware, with automatic, hardware propagation of writes to remote memories. We choose to use detailed architectural simulations since it provides many advantage at the system design stage. It offers a lot of flexibility in specifying system and especially hardware parameters, and detailed view of the system at all levels.

Second, we examine the use of hardware AU support in the communication layer in the context of emerging clusters as described in 1.2. Specifically, the following questions are addressed: (i) Given the decline of write–through capable second–level caches—and the traffic problems they raise with SMP nodes–can hardware automatic update support be provided even with write–back caches? (ii) Within the same (home–based) protocol framework, does hardware support for automatic update snooping deliver a performance benefit large enough to justify its cost, assuming both write–through capable and write–back caches? (iii) Does this hardware support for AU work well enough with the new commodity NIs like Myrinet, or does it require customized NIs like that in SHRIMP to achieve substantial benefits? That is, is AU–based SVM an argument for building customized NIs?

We use detailed architectural simulation and a wide range of real applications and computational kernels (Section 3.5.2). The simulator gives us deep visibility into all aspects of the simulated hardware and software, including queues and buffers, which is especially important for detailed system performance analysis. An associated flexible visualization tool allows us to easily examine the detailed event frequencies and contention–related bottlenecks in all parts of the node and NI on a per–node basis, and proves very important in analyzing the observed performance effects.

After discussing new design issues for protocols that support SMP nodes and use hardware support for automatic update in the presence of write–back caches, we evaluate application performance in the context of emerging clusters. We analyze how the characteristics of our ten applications and their algorithms interact with the use of SMP nodes, to see what classes of applications do and do not benefit from

SMP nodes, and determine the major bottlenecks that stand in the way of improved performance.

We find that: (i) the performance of both home-based protocols (AURC and HLRC) improves substantially with the use of SMP nodes in five of the ten applications. In three applications there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ across all–software and automatic update protocols, with the latter performing worse with SMPs than with uniprocessors; (ii) AU support for SMP nodes with write–through caches does significantly improve performance of some irregular applications, especially when diff–related costs dilate critical sections and increase serialization, but it can also hurt substantially in some cases; (iii) AU support with write–back caches solves the problems caused by the increased traffic and performs best, but is very intrusive into the underlying node; (iv) overall, the benefits of AU support are quite limited with commodity NIs with their higher occupancies per packet, and NIs customized for AU packet generation may be important for using AU effectively.

3.1 Lazy Release Consistency

Lazy Release Consistency (LRC) [52] is a particular implementation of release consistency (RC) [36, 33, 34]. RC is a memory consistency model that guarantees memory consistency only at synchronization points. These are marked as acquire or release operations. In implementations of an eager variation of release consistency the updates to shared data are performed globally at each release operation. LRC is a relaxed implementation of RC which further reduces the read–write false sharing by postponing the coherence actions from the release to the next related acquire operation. To implement this relaxation, the LRC protocol uses *time–stamps* to establish the happened–before ordering between causally–related events [52]. To reduce the impact of write–write false sharing LRC has most commonly been used with a software or hardware supported multiple–writer scheme. The first software–based multiple writer scheme was used in the TreadMarks system [51, 52]. In this scheme, every writer records any changes it makes to a shared page during each time interval. When a processor first writes a page during a new interval it saves a copy of the page, called a *twin*, before writing to it. When a release synchronization operation ends the interval, the processor compares the current (dirty) copy of the page with the (clean) twin to detect modifications and consequently records these in a structure called a *diff*. The LRC protocol may create diffs either eagerly at the end of each interval or on demand in a lazy manner. On an acquire operation, the requesting processor invalidates all pages by consulting the information about modified pages; this information is received in conjunction with the lock from the last owner of a lock. Consequently, the next access to an invalidated page causes a page fault. In the style of the LRC protocol used in TreadMarks [51] the page fault handler collects all the diffs for the page from the writers and applies them locally in the proper causal order to reconstitute the page coherently.

3.2 Home–based LRC Protocols

Home–based LRC (HLRC) protocols [44, 45, 100] are much like the protocol described above, except that propagation of updates is managed differently. Instead of writers retaining their diffs and the faulting processor obtaining the diffs from all the writers upon a fault, the idea here is for writers to propagate their changes to a designated home copy of the page before a release operation. The writes from different processors are merged into the home copy, which is therefore always up to date according to the consistency model. On a page fault, the faulting processor simply obtains a copy of the page from the home. As a result of fetching the whole page rather than diffs, this protocol may end up fetching a greater amount of data in some cases, but it will reduce the number of messages since the data have to be fetched from only one node.

The all–software implementation of HLRC [100] uses software write detection and diff–based write propagation schemes similar to TreadMards [51]. Diffs are computed at the end of each time interval for all pages updated in that interval. Once created, diffs are eagerly transferred to the home nodes of the pages, where they are immediately applied. Therefore, diffs are transient, both at the writer nodes and at the home nodes. Writers can discard their diffs as soon as they are dispatched, greatly reducing the memory requirements of the protocol. Home nodes apply arriving diffs to the relevant pages as soon as they arrive, and immediately discard them too. Later, during a page fault, following a coherence invalidation, the faulting node fetches the correct version of a whole page from the home node.

Some recent network interfaces also provide hardware support for the propagation of writes at fine granularity (a word or a cache line, say) to a remotely mapped page of memory [16, 50, 37]. This facility can be used to accelerate home–based protocols by eliminating the need for diffs, leading to a protocol called automatic update release consistency or AURC [46]. In fact, hardware support for automatic update inspired the design of home–based protocols. When a processor writes to pages that are remotely mapped (i.e. writes to a page whose home memory is remote), these writes are automatically propagated in hardware and merged into the home page, which is thus always kept up to date. At a release, a processor simply needs to ensure that its writes to remote pages so far have been flushed to the home. At a page fault, a processor simply fetches the page from the home as before.

The advantages of home–based protocols can be summarized as follows: accesses to pages on their home nodes require no diffs and cause no network traffic even if the pages have been written to by other processors, non–home nodes can always bring their shared pages up–to–date with a single round–trip rather than potentially several messages, and protocol data and messages are much smaller than under standard LRC. Studies on different platforms have indicated that home–based protocols outperform traditional LRC implementations, at least on the platform and applications tested, and also incur much smaller memory due to the transient nature of diffs [44, 46, 45, 100].

Having understood the basic protocol ideas, let us proceed to examining how and how well the protocols can be used to extend a coherent shared address space in software across SMP nodes.

3.3 Extending Home–based Protocols to SMP Nodes

Implementing the HLRC protocol on SMPs (HLRC-SMP) requires several non–trivial changes due to the interactions of hardware–coherent intra–node shared memory with the software–coherent inter–node sharing. The main goal is to take advantage of the hardware cache coherence within each node and design protocols that incur minimal overhead for local operations. In this section we discuss some of the critical issues related to the efficiency of an SVM implementation for SMPs. Section 3.4 describes the specific choices made in our implementation.

3.3.1 Sharing models

In this subsection we present the high–level issues in designing a protocol for a system with SMP nodes. The key issue is which data sharing model is appropriate for such a system:

Shared–nothing model: The uniprocessor implementations can be made to work with SMP nodes with virtually no modifications, if the protocol treats each processor as if it were a separate node. The processors do not share any application or protocol data and the hardware shared memory in a node is used merely as a fast communication layer for message passing. However, such a model does not leverage the cache–coherent shared memory provided within the SMP.

Shared–everything (thread) model: At the other extreme we consider a model where all the processors within a given node share the same view of both the application data and all the data structures used by the SVM system. In such a model the node would appear to contain a single processor to the outside world. When coherence actions are performed they apply to all the processors within the node. For example when a processor acquires a lock from a remote node and invalidates pages, the page invalidations are performed for all the processors in this node. This is of course conservative, since the other processors in the node do not need to see these invalidations yet according to the consistency model. However, acquires within a node (local acquires) will require almost no protocol overhead (e.g., no page invalidations), since the updates performed locally will already be made available by the intra–node hardware cache–coherence. Since all the processes within a node always have the same state for any given page, one page table suffices to keep the necessary state information for all the processes. This is akin to the thread model of computation within the node, where all threads have the same view of the shared address space and use one page table to maintain the page state information. The first time a thread modifies data on a shared page, a twin page is created to hold the original version of the page. At release points, the two versions of each modified page are compared and the modifications to data are sent in the forms of diffs to the home of each shared page. Thus, the propagation of diffs occurs at barrier synchronization and remote lock acquires (which can be seen as a delayed lock release). As a result, lock releases are very cheap, except when there is a remote lock request waiting for this lock.

Unlike the previous model, this one does utilize the hardware cache–coherence to share application data within the SMP and also to share a number of data structures required by the SVM system itself. However, applying invalidations unnecessarily to all processors in an SMP node can degrade performance significantly. In particular, the effects of page–level false sharing in this *eager invalidation* scheme can be large for irregular applications that exhibit relatively fine–grain accesses to shared data, resulting in a large number of page faults and page fetches.

A hybrid lazy (process) model: The shared–everything model utilizes the SMP hardware as much as possible, but it is the shared–nothing model in which coherence information is propagated only when absolutely necessary (i.e. as lazily as possible). To provide both these desirable features, we propose and implement a scheme with *lazy invalidations*. In this scheme all processors within a node share all the application data and a number of data structures used by the system. However, each process has its own page–table, and a given page in the system may have different states for different processes. Now, during a remote lock acquire, invalidations are performed *only* for the acquiring process (this will help to make the acquire faster). However a local lock acquire will now require invalidations to be performed. Thus, a local lock acquire will be more expensive than the shared–everything scheme, but we must design it to be much less expensive than the shared–nothing scheme. Figures 3.1 and 3.2 show how the coherence actions are performed at different times, in the case where all the processors in the node acquire the lock.

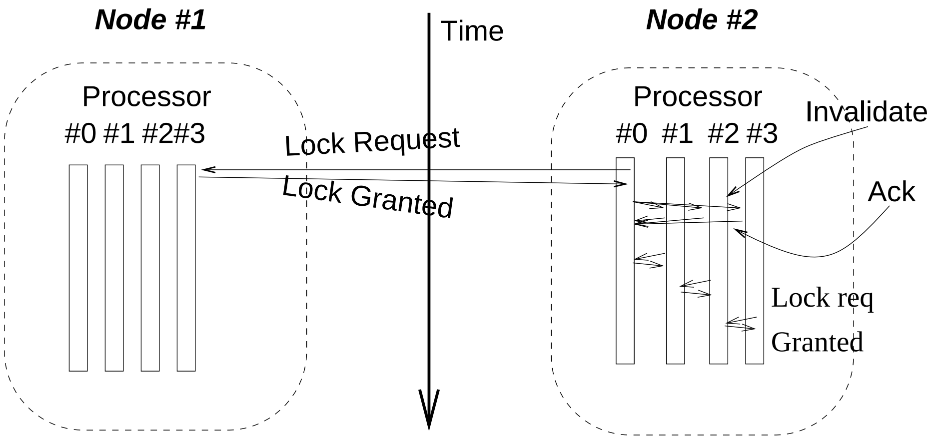

Figure 3.1: Eager invalidation schemes.

Similarly to the previous scheme, we perform diffs at barriers and incoming remote lock requests, as well as lock releases when there is an outstanding remote request for this lock. Barrier operations are similar in this and the shared–everything schemes.

3.3.2 Translation–lookaside buffer coherence

Previous studies [29] presented Translation–Lookaside Buffer (TLB) synchronization as a major obstacle for an SVM implementation to achieve good performance on a cluster of SMPs. TLB synchronization or TLB shootdown are terms used for the global operation (within an SMP) of flushing the TLB of all the processors within an SMP at various protocol operations. When exactly the TLBs of other processors need to be invalidated depends on the operating system (and the functionality that certain operating system calls provide) and also the protocol that is used. The designer needs to worry about what happens when processes migrate, when the protection of pages is modified and when data is updated with protocol operations.

At process migration, if stale TLB entries exist in another processor's TLB, the process that migrates may not see the most updated version of data. This can happen

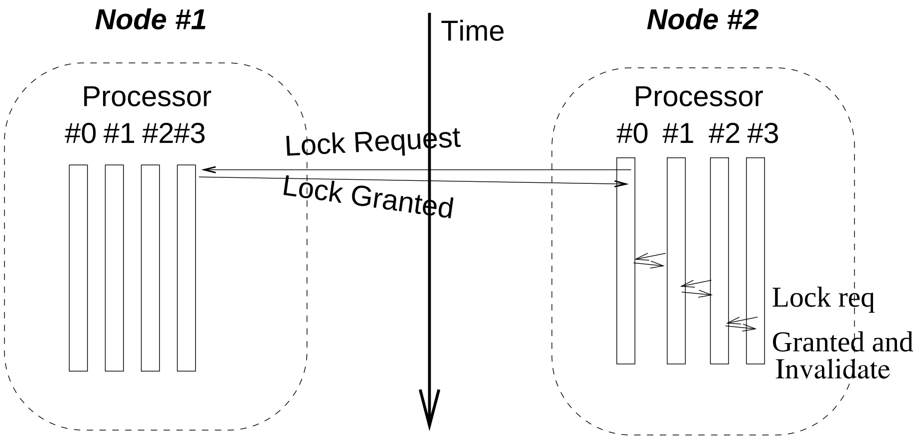

Figure 3.2: Lazy invalidation schemes.

if the TLB is treated as a non-coherent cache for the processor page table. Since the Pentium family of processors does not support entries of multiple processes in the same TLB, TLBs are flushed on every context–switch. Hence, process migration does not pose a problem on this architecture.

When the protection of a page changes, depending on whether processes or threads are used the TLBs of all processors in the node may need to be flushed. For instance, in the eager invalidation scheme, when one processor performs a change to the page table, all compute threads need to see the changes so the TLBs of all processors need to be flushed (entirely or partially depending on the processor) either by the protocol or the operating system. The designer has to take into account the specific actions taken by the operating system and adapt the protocol to handle page invalidations appropriately.

The cost of flushing TLBs of other processors may vary significantly depending on whether it is performed in the operating system or in the protocol, since inefficient user signals may have to be used in the latter case.

3.3.3 Synchronization

Barriers may create hot spots in the network if they are not implemented carefully. In an SMP configuration, two–level hierarchical barriers not only reduce hot spots, but reduce the number of messages exchanged as well. The lower level is concerned with intra–node synchronization that does not involve any messages at all, and the higher level with inter–node synchronization, which is achieved by exchanging messages. Hierarchical barriers in an SMP configuration match the underlying architecture well.

Hierarchical barriers can be either centralized or all–to–all. In centralized barriers, synchronization is implemented (in the absence of hardware support) with messages that are sent by each SMP node to a barrier master. The barrier master, gathers the control information and distributes it to all the nodes after they have reached the barrier. In all–to–all barriers, also called n^2–barriers, each node sends synchronization and control messages to every node in the system. The tradeoff between centralized and all–to–all barriers is mainly the number of messages that are exchanged and the amount of serialization introduced. Centralized barriers use fewer messages, but incur more serialization at the barrier master.

If centralized barriers are used, another important tradeoff is the amount of processing needed at the barrier master. The gathered control information (invalidations and time–stamps) can either be processed locally in the barrier master first and then sent to each node only as necessary, or it can be sent to all nodes and then the appropriate information can be extracted locally. The first approach reduces the size of messages that are sent but exhibits higher serialization at the barrier master. The second approach uses bigger messages but exhibits higher parallelism.

Locks within an SMP node need not exchange messages. This makes local lock acquires very cheap. Depending on the invalidation scheme used (as discussed above) local lock acquires can be as cheap as a few memory references.

3.3.4 Protocol handling

In all SVM implementations remote requests sent over the network need to be serviced asynchronously. On a uniprocessor node there is little choice in this regard in the absence of dedicated hardware support; the processor must be interrupted or it must somehow poll periodically. However, with SMP nodes there are a number of choices. The two basic ideas are either to dedicate a processor within the SMP to handle network requests exclusively (by polling) or to handle the requests by interrupting one of the computation processors.

A dedicated processor implementation helps to avoid interrupts. However, this choice wastes a valuable resource, by reducing the number of compute processors. Our experiments as well as other work [49, 31] show that this dedicated processor is mostly idle, since actual protocol processing overhead is not very high.

If we do not devote a processor to protocol handling and we use the compute processors to handle requests, then we could either statically assign one compute processor for this purpose or we could perform a round–robin assignment as the requests arrive. To reduce interrupts we can instruct idle compute processors to poll for requests and interrupt a random compute processor only when there are no idle processors [49].

On a real system some choices may be difficult or too expensive to perform due to architectural and operating system limitations. For instance, Linux 2.0.x delivers incoming interrupts only to processor 0, and it is not possible to distribute interrupts among processors within an SMP.

3.3.5 Protocol optimizations

Finally, several system aspects can influence performance substantially and change the tradeoffs in protocol design. These include various architectural and operating system costs, e.g., interrupts, network latency and bandwidth, etc. These issues need to be taken into account both at the design and the implementation of a protocol.

3.4 Protocol Implementation

In this section we present several implementation issues and details of the all–software HLRC-SMP protocol, which follows the hybrid sharing model as described above. For the most part, these apply AURC-SMP and WBAURC-SMP as well.

3.4.1 Data structures and synchronization intervals

To illustrate the data structures, we first need to define some key terms. The time during the actual execution of a parallel program is broken into *intervals*. With uniprocessor nodes, intervals are maintained on a per–process basis. An interval in a process's execution is the time between two consecutive releases by the process. These intervals are numbered in a monotonically increasing sequence. Each process maintains a vector called the *update–list*, which records all the pages that have been modified (by this process) in the current interval. Intervals are ended generally at release points in lock acquires, releases and barriers.

When we end an interval, this update–list is placed in a data structure called the *bins*. This is the key data structure used by the SMP protocol. In our SMP protocol for the simulator we use a single column for each processor in the system. Intervals are therefore maintained per processor. This allows for finer grain intervals than the case where the intervals are maintained per node.

Each SMP node contains a copy of the entire bins data structure. The entry in a given bin (say interval 2, processor 1) will be the same in every node's copy of the bins data structure. However, for a particular node A, when the entry for a processor in some other node B for a particular interval will appear in A's bins depends on when A does a causally related acquire from a processor in B and receives them. Essentially the bins inform each processor which pages were modified in relation to which synchronization operation, and in which order. In conjunction with the time–stamps tell the processor which pages need to be invalidated at a synchronization operation to maintain consistency. This information is also used to infer version information about the pages, so that when pages are fetched from the home node, we know which versions to demand.

In addition to this data structure, each process in a node uses a simple data structure called the *view vector,* which can also be seen as a vector time–stamp. In the simulator this is an array of integers with size equal to the number of processors in the system. Essentially this is the *view of the world* that a process has. This vector maintains the information regarding what portion or height of the bins for different nodes has been seen (i.e. the invalidations corresponding to those intervals from different nodes have been performed) by this particular process. When one process fetches new bin information from another node on an acquire, the new information is available to all processes in it's SMP node. However, this and the other processes do not *act* on all this information immediately. Processes only invalidate pages for their own acquires, and only the pages (i.e. the entries in the bins) that they are required to see, as dictated by the causal ordering of intervals. The view vectors are used to maintain this ordering and indicate which intervals have to be seen by each process at any point during its execution. Figure 3.3 shows how this works. Essentially, the bins data structure in each node, together with the view vectors maintains the execution graph for all operations in the system.

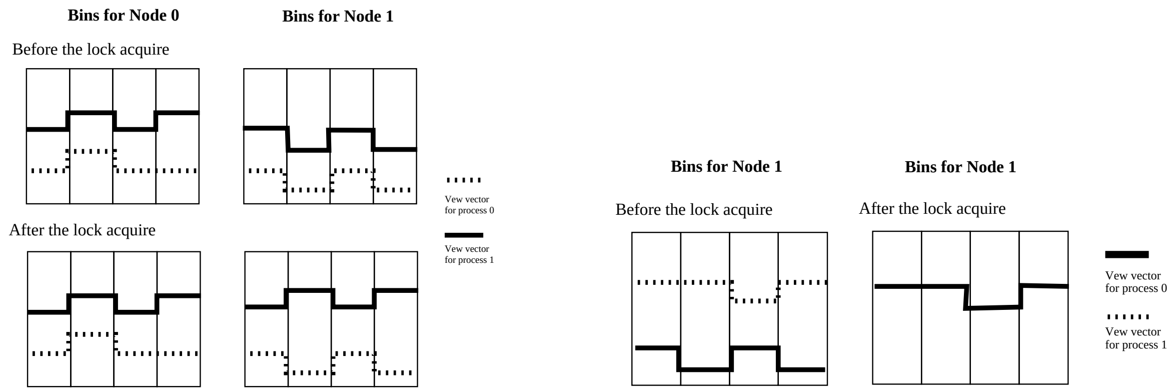

Figure 3.3: View synchronization during remote (left) and local (right) acquire operations.

During a remote lock acquire operation (when the requested lock is available at a remote node), the requesting process sends over its view vector and a vector that indicates what bins are currently present at this node. Any portions of the bins that are not available at the requester are sent back in the form of *write notices,* along with the view vector of the releaser of the lock. The requester then matches its view vector with the lock releaser's view vector (as it was at the time of the lock release operation), and invalidates, for itself only, all the pages indicated in the bins that are seen by the releaser's view vector but not its own. This operation is illustrated in Figure 3.3. In this example process 1 on node 1 is acquiring a lock which was previously released by process 1 on node 0.

3.4.2 Translation–lookaside buffer coherence

In our protocol, there is no need to flush the TLBs explicitly, since this is taken care of by the x86 architecture and the Linux operating system at context switches and page invalidations. The x86 architecture flushes the TLB at context switches and Linux flushes the processor TLB when the state of a page is modified.

3.4.3 Synchronization

The algorithm that implements locks in the base HLRC-SMP protocol is used in many other protocols as well. Every lock is statically assigned a home. When a process needs to acquire a lock it sends a message to the home of the lock. The home forwards the message to the last owner and the owner releases the lock to the requester. The requests at the home and at the last owner are both handled using interrupts. The home of each lock is thus used to maintain a distributed list of locks. The last owner keeps the lock until another processor acquires the lock.

When the owner passes the lock along to another process it also sends along with the lock the invalidations that the requester needs to have to maintain the release–consistent view of the shared memory. Thus, since the base protocol uses lazy propagation of invalidations, when the protocol handler services a remote acquire it sends to the requester both the lock (mutual exclusion part) and the page invalidations (coherence information).

The scheme we use for locking allows a local lock acquire operation to be completely local. All that is required is the matching of the requester's view vector with the releaser's, hence there is some protocol processing with only necessary invalidations taking place. As we can see in Figure 3.3, this operation involves only the local–node's bins, and no data transfers across nodes are involved.

We implemented both all–to–all and centralized barriers and we found that for the systems and scale under consideration there is no substantial difference in performance. The protocol processing that is performed at each processor includes synchronizing the vector with the most updated view from each processor in the system, updating page time–stamps to the latest versions that are required, and invalidating the necessary pages.

Synchronization within nodes does not use interrupts, which are needed only to service remote page fetch and synchronization requests. These remote requests are handled by a statically assigned processor in each node, as discussed earlier.

3.4.4 Protocol Handling

The scheme used for protocol handling is asynchronous processing of the incoming requests with interrupt handlers at user–space. Since Linux 2.0.x delivers all incoming interrupts to processor 0, this processor is used to handle the remote protocol requests.

3.4.5 Performing diffs

Performing diffs occupies a non–trivial fraction of the time spent in the SVM system. A simple way to update home pages is to diff at every release point in the protocol. This makes diffing synchronous to protocol processing in each process and reduces the possibility for race conditions. However, it leads to unnecessary diffing, e.g. when a lock is repeatedly acquired by the same processor. To reduce the amount of diffing in the system, we implement a strategy where we diff only at a barriers and when a lock is granted to a remote node. This lazy diffing scheme helps to significantly reduce the cost of local lock releases. However, the scheme does introduce a certain amount of complexity. Attempting to fetch a page for which we have outstanding updates waiting to be diffed will result in a loss of these updates. This is avoided by marking each modified page as *dirty*. The page remains *dirty* even after a release, until it is diffed, which will cause all fetches of that page from the home to be safe page fetches as described next.

3.4.6 Page fetches

Finally, let us see how data is fetched when a more recent version of a page is needed. The home node of a page always has the most current version of the page. However, updates from other nodes in the system may be in flight (in the form of diffs), so it is desirable when requesting a page to specify the version that is absolutely necessary. To achieve this we use a system of *lock time–stamps* and *flush time–stamps*. Each page at the home has associated with it a flush time–stamp that indicates what is the latest interval for each node for which the updates are currently available at the home. One may think of this vector as a *version* of the page. On a page fault, the requester sends to the home a lock time–stamp, corresponding to its last lock acquire, which indicates to the home what the flush time–stamp of the page should at least be, in order to ensure that all relevant changes to the page by other processors are in place. If the flush time–stamp is less than the lock time–stamp, then diffs must be on their way so the home waits for the diffs to arrive before satisfying the page request. Essentially, page time–stamps specify the version of data that are requested or updated. Thus, no particular ordering, besides FIFO delivery between two processes, is required in the network for data request and update messages from different processors; they can arrive at any order, and they will be serviced in the right way. This means that HLRC-SMP does not require any global ordering guarantees to be met by the communication layer for data request and update messages.

During the page fetch operation, there is one major complication. If there are outstanding updates to the local copy of a page (which have not been communicated to the home via diffs) made by the requesting or some other local processor in the in the SMP, fetching the page in its entirety will overwrite these updates to the page. To avoid this problem we use a *dirty bit* which detects this scenario. This bit is set on the first write by any process in the SMP to that page. It is cleared when a diff is performed on the page and there are currently no active writers to the page within the node. When a page is observed to be dirty we perform a *safe page fetch*. Essentially a safe page fetch diffs the page returned by the home with the local twin of the page and applies these updates to the current local page as well as the current twin for this page. This scheme updates the local twin and page with the newer version of the data from the home without loosing any local updates. The scheme is similar to the *2–way diffing* used in Cashmere–2L [91].

3.5 Evaluation Methodology

In this section we present the simulation environment that we use in in Chapters 3, and 4 and the applications we use to evaluate system performance.

3.5.1 Simulation environment

The simulation environment we use is built on top of Augmint [81], an execution driven simulator using the *x86* instruction set and runs on *x86* systems.

The simulator consists of two levels as shown in Figure 3.4. The lowest layer implements a detailed architectural simulator for the architecture under consideration and the higher layer implements the different SVM protocols. The applications that are run in the simulator are the same ones that are run in the actual system.

The simulated architecture (Figure 3.5) assumes a cluster of c–processor SMPs connected with a commodity interconnect like Myrinet [17]. Contention is modeled at all levels except in the network links and switches themselves.

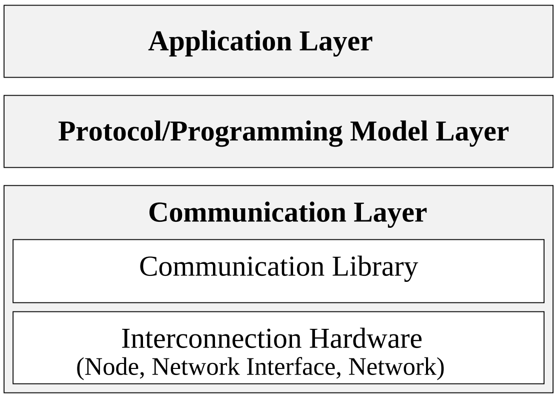

Figure 3.4: The layers that affect the end application performance in software shared memory.

The processor is roughly modeled after the Intel P6 processor. It has a P6–like instruction set, and is assumed to be a 1 IPC processor. The data cache hierarchy consists of an 8–KByte, first–level, direct–mapped, write–through cache and a 512–KByte, second–level, two–way, set–associative cache, each with a line size of 32 Bytes. The write buffer [90] has 26 entries, 1 cache line wide each, and a retire–at–4 policy. Write buffer stalls are simulated. The read hit cost is one cycle if satisfied in the write buffer and first level cache, and 10 cycles if satisfied in the second–level cache. The memory subsystem is fully pipelined.

The memory bus is split–transaction, 64 bits wide, with a clock cycle 4x slower than the processor clock. Arbitration takes one bus cycle, and the priorities are, in decreasing order: second level cache, write buffer, memory, incoming path of the network interface, outgoing path of network interface. The I/O bus is 32 bits wide and has a clock speed half that of the memory bus. The *relative* bus bandwidths and processor speed match modern systems such as the one we use in the real implementation. If we assume that the processor has a clock of 200MHz, the memory and I/O buses are 400 MBytes/s and 100 MBytes/s respectively.

Each network interface (NI) has two 1 MByte memory queues for incoming and

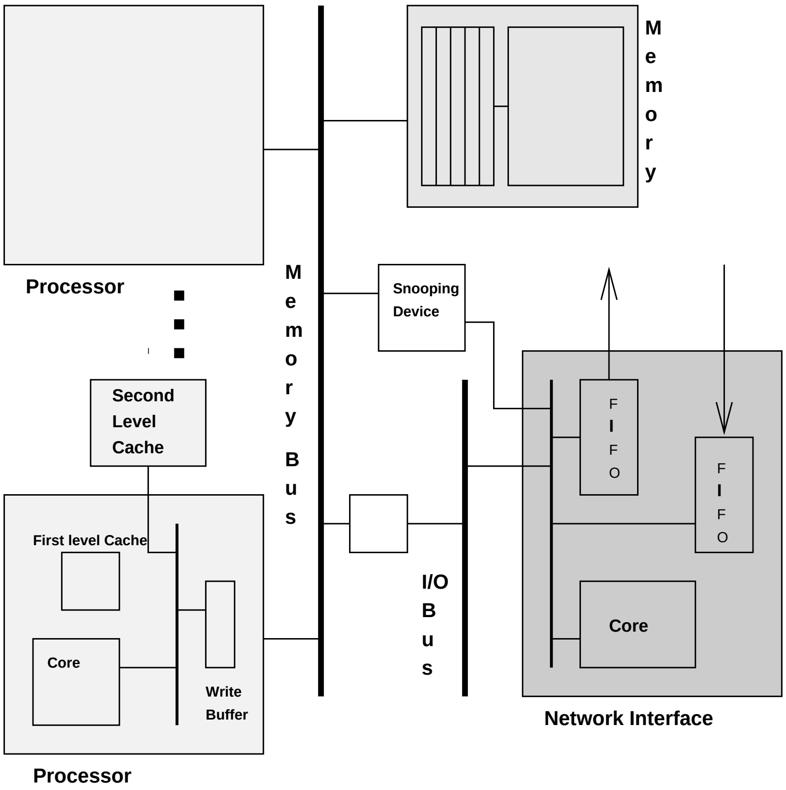

Figure 3.5: Simulated node architecture.

outgoing packets. Network links operate at processor speed and are 16 bits wide. We assume a fast messaging system [24] that supports explicit messages. Initiating a message (host overhead) takes on the order of hundreds of I/O bus cycles. A snooping device on the memory bus forwards AU traffic to the NI [16]. If the network queues fill, the NI interrupts the main processor and delays it for a fixed number of cycles to allow queues to drain. The NI sets up network packets, which incurs a cost per packet (about 1000 host processor cycles). Packets are delivered directly to memory, without processor intervention at the receive side. The packet size in the network is 256 bytes for DMA transfers and 4 bytes (1 word) for AU transfers. AU writes within a cache line are combined in the NI to reduce the number of packets.

Similarly to what happens in Linux, incoming interrupts are delivered to processor

0 in each node. More complicated schemes (i.e. round robin, random assignment) that result in better load balance in interrupt handling can be used if the operating system provides the necessary support. These schemes however, may increase the cost of delivering interrupts. Issuing an interprocessor interrupt costs 500 processor cycles, and invoking the handler is another 500 cycles. This is aggressive compared to what current operating systems provide, but is implementable [92] and prevents interrupt cost from swamping out the effects of other system parameters [12].

The page size is 4 KBytes, and the cost to access the TLB from a handler running in the kernel is 50 processor cycles. Each protocol handler is charged a cost depending on the work it does. The cost of creating and applying a diff in HLRC-SMP is computed by adding 10 cycles for every word that needs to be compared and 10 additional cycles for each word included in the diff.

In setting the simulator parameters our main goal was not so much to exactly match the absolute values of the parameters in real systems but to maintain the important relations among them. Since the processor is less aggressive than the latest generation of processors (single issue at 200MHz versus multiple issue at 200 or more MHz), we scaled down the values that affect the performance of the memory subsystem and the NI as well, and we use slower memory and I/O buses.

While we try to be very close to a realistic system in our simulations, we sometimes set parameters to avoid certain artifactual limitations of the real system. For instance the interrupt cost in the simulator is lower than on todays systems. Moreover the simulator does not deal with interrupt handler scheduling artifacts that arise in the real implementation. This approach allows for fewer artifacts in the evaluation, but it does lead to somewhat larger speedups than in the implementation (the breakdowns of execution time into various components validate quite well given these differences, Section 6.1). The problem sizes that are simulated, while realistic for each application, are at the small end of what might be run in practice, and this needs to be taken into account when conclusions are drawn.

The programming model provided by the simulator is the ANL macros. The simulator performs first touch allocation for shared data pages. However, to avoid improper data allocation, shared data are allocated in each application in accordance with SPLASH-2 [97] guidelines. Statistics are similarly reset in accordance with SPLASH-2 guidelines.

The simulator provides detailed statistics about all events in hardware, as well as statistics that help identify contention in the various components of the system. For instance, the simulator can report packet wait times to various queues in the system, per packet type. One limitation of the simulator is that protocol handlers can not be simulated since the simulator itself is not multi–threaded. Handlers are ascribed a cost depending on the number of instructions they execute. A flexible visualization tool for these detailed statistics (frequencies, times, and contention) proved very valuable in analyzing the performance effects we discuss.

3.5.2 Applications

In our evaluation we use the SPLASH-2 [97, 46] application suite. We will now briefly describe the basic characteristics of each application. The applications can be divided in two groups, regular and irregular.

Regular applications

The applications in this category are FFT, LU and Ocean. Their common characteristic is that they are optimized to be single–writer applications; a given word of data is written only by the processor to which it is assigned. Given appropriate data structures they are single–writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are mostly local. The applications have different inherent and induced communication patterns [46, 97], which affect their performance and the impact on SMP nodes.

FFT [2, 98]: The FFT kernel is a complex 1–D version of the radix–$\sqrt{n}$ six–step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed and another n complex data points referred to as the roots of unity. Both sets of data are organized as matrices, which are partitioned so that every processor is assigned a contiguous set of $\sqrt{n}/p$ rows that are allocated in its local memory. Communication occurs in three matrix transpose steps, which require all–to–all interprocessor communication. Every processor transposes a contiguous sub–matrix of $\sqrt{n}/p$–by–$\sqrt{n}/p$ elements from every other processor to itself—thus reading remote data and writing local data—and transposes one sub–matrix locally. The transposes are blocked to exploit cache line reuse. To avoid memory hot–spotting, sub–matrices are communicated in a staggered fashion, with processor i first transposing a sub–matrix from processor $i+1$, then one from processor $i+2$, etc.

LU: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense n–by–n matrix A is divided into an N–by–N array of B–by–B blocks ($n = NB$) to exploit temporal locality on sub–matrix elements. To reduce communication, block ownership is assigned using a 2–D scatter decomposition, with blocks being updated by the processors that own them. The block size B should be large enough to keep the cache miss rate low, and small enough to maintain good load balance. Fairly small block sizes (B=8 or B=16) strike a good balance in practice. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them [98]. We use two versions of LU that differ in their organization of the matrix data structure. The contiguous version of LU uses a four–dimensional array to represent the two–dimensional matrix, so that a block is contiguous in the virtual address space. It then allocates on each page the data of only one processor. The non-contiguous version uses a two–dimensional array to represent the matrix, so that successive sub–rows of a block are not contiguous with one another in the address space. In this version, data written by multiple processors span a page. LU exhibits a very small communication to computation ratio but is inherently imbalanced.

Ocean [18, 84]: The Ocean application studies large–scale ocean movements based on eddy and boundary currents. It partitions the grids into square or row sub–grids rather than groups of columns to improve the communication to computation ratio. Each 2–D grid is represented as a 4–D array in the contiguous version, with all sub–grids allocated contiguously and locally in the nodes that own them. The equation solver used is a red–black, W–cycle multi–grid solver. The communication pattern in the Ocean simulation application is largely nearest–neighbor and iterative on a regular grid. We use both the contiguous (4–D array) and non-contiguous (2–D array) versions of Ocean. Also in the non–contiguous version we use either square sub–grids or sub-grids that consist of contiguous rows (Ocean-rowwise).

Irregular applications

The irregular applications in our suite are Barnes, a hierarchical N–body simulation; Radix, an integer sorting program; Raytrace, a ray tracing application from computer graphics; Volrend, a volume rendering application; and Water, a molecular dynamics simulation of water molecules in liquid state.

Barnes [3, 40, 86]: The Barnes application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time–steps, using the Barnes–Hut hierarchical N–body method. It represents the computational domain as an octree with leaves containing information about the bodies and internal nodes representing space cells. Most of the time is spent in partial traversals of the octree (one traversal per body) to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance. Access patterns are irregular and fine–grained. We use two versions of Barnes, which differ in how the shared octree is built and managed across time–steps. The first version (Barnes-rebuild) builds the tree from scratch after each computation phase. The second version, Barnes-spatial [47], is optimized for SVM implementations—in which synchronization is expensive—and it avoids locking as much as possible. It uses a different tree–building algorithm, where each processor first builds its own partial tree, and all partial trees are merged to the global tree after each computation phase.

Radix [30]: The integer radix sort kernel is based on the method described in [13]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all–to–all, irregular communication.

The permutation is inherently a sender–determined one, so keys are communicated through scattered, irregular writes to remotely allocated data [42, 98]. We use the original version of Radix as well as a modified version (Radix-local) that exhibits less scattered accesses to remote memory.

Raytrace [83]: This application renders a three–dimensional scene using ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene, and early ray termination and anti–aliasing are implemented, although anti–aliasing is not used in this study. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing for load balancing. The major data structures represent rays, ray trees, the hierarchical uniform grid, task queues, and the primitives that describe the scene. The data access patterns are highly unpredictable in this application. The version we use is modified from the SPLASH–2 version [97] to run more efficiently on SVM systems. A global lock that was not necessary was removed, and task queues are implemented better for SVM and SMPs [47]. Inherent communication is small. We present results only for the SMP protocols due to simulation cycle limitations.

Volrend [65]: This application renders a three–dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace. The main data structures are the voxels, octree, and pixels. Data accesses are input–dependent and irregular, and no attempt is made at intelligent data distribution. The version we use [47] is slightly modified from the SPLASH–2 version [97]; task are stolen first from processors in the local node and them from remote nodes. Inherent communication volume in Volrend is small.

Water [85]: This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed every time–step, and a predictor–corrector method is used to integrate the motion of the water molecules over time. We use two versions of Water, Water-nsquared and Water-spatial. The first uses an $O(n^2)$ algorithm to compute the forces, while the second computes the forces approximately using a fixed cutoff radius, resulting in an $O(n)$ algorithm. Water-nsquared can be categorized as a regular application, but we put it here to ease the comparison with Water-spatial. In both versions, updates are accumulated locally between iterations and performed at once at the end of each iteration. The inherent communication to computation ratio is small.

3.5.3 Application statistics

Tables 3.1, and 3.2 and Figures 3.6–3.10 can be used to characterize the applications. Table 3.2 presents counts of protocol events for each application, for 1, 4 and 8 processors per node (16 processors total in all cases). From these statistics we see that the applications exhibit very different behavior with respect to the communication subsystem, covering a wide range of application behaviors. We can use these statistics

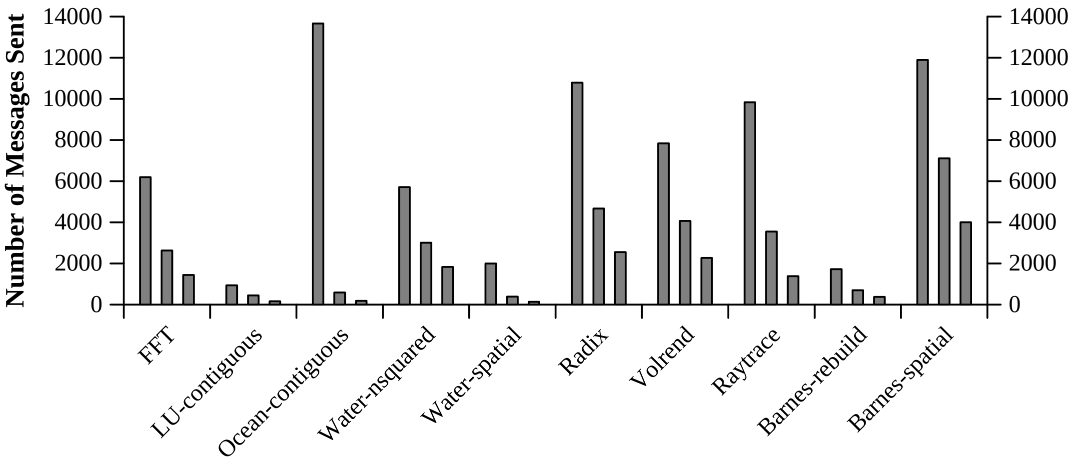

Figure 3.6: Number of messages sent per processor for each application for 1 (left), 4 (middle) and 8 (right) processors per node, for a total of 16 processors.

to categorize the applications in terms of the communication they exhibit. Both the number of messages and MBytes of data exchanged are important to performance; figure 3.10 shows for each application the geometric mean of the normalized value for the numbers of messages and the amount of traffic. The normalization is done per processor per–10^7 cycles of application compute time and then the results are averages across all processors for each application. If we use this geometric mean, which captures multiplicative effects, as a metric, then we can divide the applications in three groups. In the first group belong Barnes-rebuild, FFT and Radix that exhibit a lot of communication. In the second group belong Water-nsquared and Volrend that exhibit less communication and in the third group the rest of the applications, LU, Ocean, Water-spatial, Raytrace and Barnes-spatial that exhibit very little communication. It is important to note that this categorization holds for the configurations with four and eight processors per node. In the configuration with uniprocessor nodes, Ocean also belongs in the category of applications that exhibit a lot of communication.

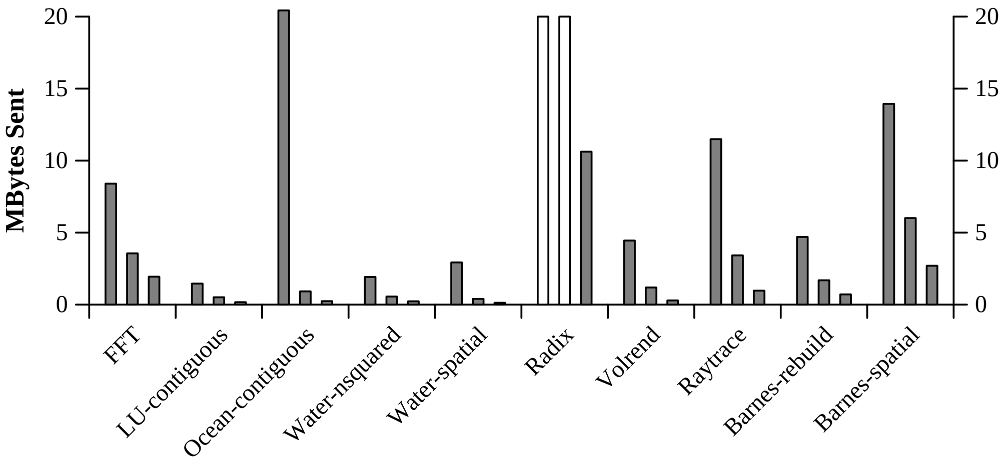

Figure 3.7: Amount of data sent per processor for each application for 1 (left), 4 (middle) and 8 (right) processors per node, for a total of 16 processors.

In Ocean the nearest neighbor communication pattern results in high communication volume when there is one processor per node and all communication for shared data is external to the node, and in low communication volumes with four or eight processors per node, with most of the communication being local in each node.

Appl.	Prob. Size	Page Faults			Page Fetches			Local Locks			Remote Locks			Bar
		1	4	8	1	4	8	1	4	8	1	4	8	1,4,8
FFT	20	2082	1320	1417	2062	876	480	0	0	0	0	0	0	6
LU-cont	512	283	197	167	250	122	41	0	1	1	1	0	0	67
Ocean-cont	514	4467	811	713	4463	172	50	0	5	9	15	10	6	90
Water-nsq	512	399	127	46	393	110	42	0	694	911	1171	477	260	19
Water-spa	512	514	113	48	492	93	32	0	10	14	21	11	7	22
Radix	1K	2212	877	1043	2158	476	142	1	5	35	48	44	14	11
Volrend	head	1044	432	338	1041	288	64	0	288	429	441	173	39	15
Raytrace	car	2742	783	208	2742	781	207	1	68	121	149	100	41	3
Barnes-reb	8K	1905	934	501	1874	824	364	1	308	648	1152	850	498	13
Barnes-spa	8K	404	88	64	388	83	29	0	1	2	2	1	0	15

Table 3.1: Number of page faults, page fetches, local and remote lock acquires and barriers per processor for each application for 1, 4 and 8 processors per node.

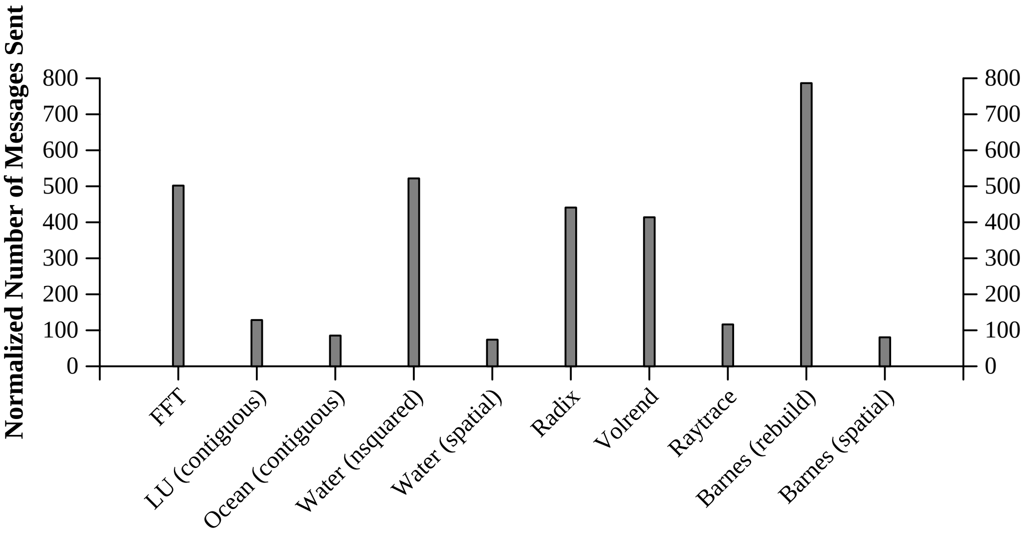

Figure 3.8: Normalized number of messages sent per processor for each application for 1 processor per node, for a total of 16 processors.

3.5.4 Metrics

While speedup is an important metric, factors unrelated to the SVM protocol can cause speedups to be high even when the protocol itself is not well suited to the application. For example, a sequential execution can perform very poorly due to the working set not fitting in the cache, a problem that may go away in a parallel execution if the application is such that the important working set diminishes as the number of processors is increased, and thus leads to very high speedups. We will see an example of this in the Ocean application. To understand how well protocols themselves perform, we use both speedups and a metric we call *protocol efficiency*.

We define protocol efficiency for a given application and protocol for N processors as:

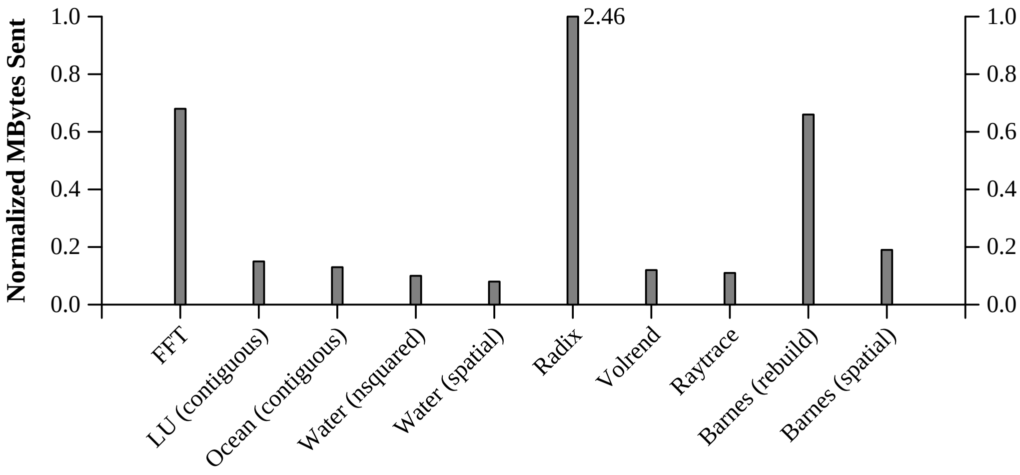

Figure 3.9: Normalized amount of data sent per processor for each application for 1 processor per node, for a total of 16 processors.

$$f_N = \frac{\sum_{0 \le i \le N}(C_i + S_i)}{\sum_{0 \le i \le N} E_i},$$

where E_i, C_i, S_i are the elapsed, compute and cache stall time of the i^{th} processor in the parallel execution. Obviously $f_N \in [0, 1]$. Note that the costs in the sequential execution are not directly involved in this definition. Assuming that a protocol is responsible for all overheads except the local cache stall time, this definition penalizes a protocol for every cost incurred. Thus, a protocol that injects overheads, besides the local cache stall time, in the elapsed time E, will have a lower efficiency factor than let's say an ideal protocol where the only overhead would be the local cache stall time (for which the efficiency factor would be $f_N = 1$).

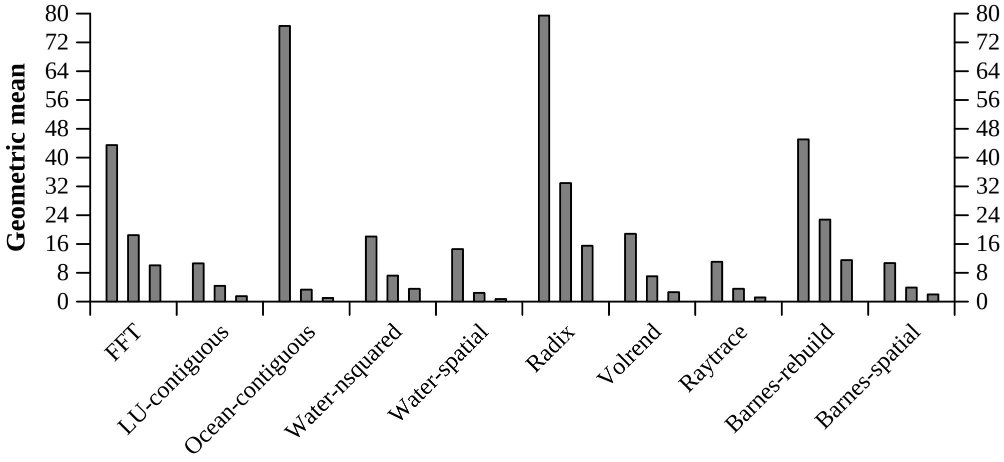

Figure 3.10: Geometric mean of the Number of messages and MBytes sent per processor per–10^7 compute cycles for each application for 1 (left), 4 (middle) and 8 (right) processors per node.

3.6 Presentation of Results

We present performance data in two main forms: tables with speedups and protocol efficiency factors in each section, and bar–graphs with per–processor breakdowns of execution time for the most interesting cases (Figures 3.15–3.14).

In the bar graphs the execution cost is divided into the following components. *Thread Compute Time* is essentially a count of the instructions executed by each thread. *CPU Stall Time* is the time the CPU is stalled on local memory references. *Thread Data Wait Time* (or page fetch time) is the time each thread is waiting for data to arrive from a remote node (page fetches). *Thread Lock Time* and *Thread Barrier Time* are the times spent in synchronization events, including idle wait time and operation overhead. *I/O Stall Time* is the time the processor spends initiating page and message transfers and waiting for the network interface queues to drain. The last component, *Handler Compute Time*, is the time spent in protocol handlers. Some of this cost is also included in other costs, for instance *Thread Data Wait Time*.

Appl.	Page Faults			Page Fetches			Local Locks			Remote Locks			Bar
	1	4	8	1	4	8	1	4	8	1	4	8	1,4,8
FFT	397.1	251.9	270.3	393.3	167.2	91.6	0.0	0.0	0.0	0.0	0.0	0.0	1.1
LU-cont	81.4	56.6	48.1	71.8	34.9	11.9	0.0	0.2	0.3	0.3	0.1	0.0	19.2
Ocean-cont	647.6	117.3	103.2	647.0	24.9	7.2	0.0	0.8	1.3	2.2	1.4	0.9	13.1
Water-nsq	69.2	22.1	8.0	68.3	19.0	7.3	0.0	120.4	158.1	203.2	82.9	45.1	3.3
Water-spa	97.9	21.4	9.2	93.8	17.7	6.0	0.0	1.8	2.6	3.9	2.2	1.4	4.2
Radix	208.8	82.7	98.4	203.7	44.9	13.4	0.1	0.4	3.3	4.5	4.1	1.3	1.0
Volrend	105.1	44.1	34.5	104.8	29.4	6.5	0.0	29.3	43.8	44.3	17.6	4.0	1.6
Raytrace	89.8	25.6	6.8	89.8	25.6	6.8	0.0	2.2	4.0	4.9	3.3	1.3	0.1
Barnes-reb	211.2	103.0	55.5	207.7	90.9	40.3	0.1	33.9	71.8	127.7	93.8	55.2	1.4
Barnes-spa	48.1	10.4	7.7	46.2	9.9	3.5	0.0	0.2	0.2	0.2	0.1	0.0	1.8

Table 3.2: Normalized number of page faults, page fetches, local and remote lock acquires and barriers per–10^7 cycles per processor for each application for 1, 4 and 8 processors per node.

To simplify the discussion, we now clarify a couple of issues up front. *Thread Data Wait Time* can be large either because many pages need to be fetched (*frequency*) or because the cost per page fetch is high due to *contention*. Imbalances in data wait time also stem from imbalances in either of these factors, and we will indicate which dominates. Lock and barrier synchronization times also have two components: the time spent waiting (for another processor to release the lock or for all processors to reach the barrier) and the time spent exchanging messages and doing protocol work (e.g., after the last processor arrives at the barrier). We will separate these out, calling the former *wait time* and the latter *protocol cost*. As we saw in Section 3.5, wait time for locks is often increased greatly in SVM systems due to page misses occurring frequently inside critical sections and increasing serialization [46], as well as to increased protocol activity at locks, which has the same effect. This makes locks much more expensive for SVM systems than for hardware–coherent systems.

The performance (cost) numbers that appear in the text are in the form of triplets of percentages or absolute values. For example, if we say that the data wait time has a cost of (10%, 45% 110%), we mean that the minimum data wait over all processors is 10% of the minimum compute time among all processors, the average 45% and the maximum 110%. Absolute values (i.e. page fetch cost in cycles, etc.) are also presented.

3.7 Uniprocessor vs. SMP Nodes

In this section we present our results that address the issues raised in the introduction with respect to extending HLRC and AURC to support SMP nodes (HLRC-SMP, AURC-SMP). We compare two system configurations for the two different protocols. The first system has uniprocessor nodes, whereas the second uses SMP nodes. In all configurations the speed of the memory and I/O buses is set to 400 MBytes/s and 100 MBytes/s respectively, and we assume a 200MHz processor. As mentioned in Chapter 3.5, these values result in a realistic commodity configuration, given the relative performance of the different components. Note that the bandwidths are the same whether the nodes are uniprocessors or multiprocessors. Tables 3.3 and 3.4 present these results.

Tables 3.5–3.8 give more detailed statistics for each application. They compare AURC and AURC-SMP, and HLRC and HLRC-SMP in more detail. For each application there are two columns. The first column presents the average percentage cost of each component of the execution time with respect to the thread compute time for the base case. The second column presents the change from the base case for the other protocol. In these tables *Barriers* refers to the total barrier cost, whereas Barrier Wait refers to the component of this cost until all processors reach the barrier (due to imbalances). Similarly *Locks* refers to the total lock cost and *Lock Wait* to the component up the point where the lock is released and can be granted to the next processor. Finally *PFetch Time* is the average cost of a page fault. Important

Application	Problem Size	Speedups			
		AURC	AURC-SMP	HLRC	HLRC-SMP
FFT	18	5.24	6.20	4.43	5.74
FFT	20	8.53	8.29	7.73	8.28
LU-contiguous	512	10.78	12.42	10.16	12.25
Ocean-contiguous	258	6.10	16.86	5.43	15.23
Ocean-contiguous	514	6.52	12.83	6.12	12.80
Barnes-rebuild	16K	5.30	6.24	2.45	3.82
Barnes-spatial	8K	12.94	12.81	11.69	12.19
Barnes-spatial	16K	13.30	12.95	10.94	11.52
Radix	1M	2.82	1.25	0.63	3.41
Raytrace	car	6.38	11.82	14.06	14.79
Volrend	head	9.14	11.95	7.86	8.81
Water-nsquared	512	9.09	9.71	8.56	8.84
Water-spatial	512	7.89	8.52	7.41	10.05

Table 3.3: Speedups for the uniprocessor and the SMP node configurations.

statistics are highlighted in the analysis of each application as well.

From Table 3.3 we can divide the applications into different classes in terms of their behavior in going from uniprocessor to SMP nodes.

3.7.1 First class

The first class is applications for which both protocols improve with SMPs. These applications are LU, Ocean-contiguous, Barnes-rebuild, Volrend and Water-spatial. We differentiate the behavior of these applications into three subgroups.

First group

The first group consists of only Ocean (Figure 3.11), which improves dramatically because of the localized, near–neighbor pattern of communication and the high amount of barrier synchronization.

Application	Problem Size	Efficiency Factors (%)			
		AURC	AURC-SMP	HLRC	HLRC-SMP
FFT	18	36	42	26	40
FFT	20	57	65	45	59
LU-contiguous	512	58	68	53	66
Ocean-contiguous	258	21	51	19	47
Ocean-contiguous	514	33	82	29	79
Barnes-rebuild	16K	33	39	15	24
Barnes-spatial	8K	86	83	74	80
Barnes-spatial	16K	89	84	70	76
Radix	1M	17	8	4	21
Raytrace	car	39	72	86	90
Volrend	head	58	76	50	56
Water-nsquared	512	58	62	55	57
Water-spatial	512	51	55	46	63

Table 3.4: Efficiency factors (as %) for the uniprocessor and the SMP configurations.

Ocean (Figure 3.11): In both AURC-SMP and HLRC-SMP data wait times are reduced significantly, compared to the uniprocessor configuration. The average page fetch cost however, is increased by about 100% and 30% in AURC-SMP and HLRC-SMP, respectively, because of the larger contention in the memory bus and network interface. The reduction in data wait time comes from the sharing pattern in Ocean. The communication pattern is nearest neighbor, so if the processes assigned to an

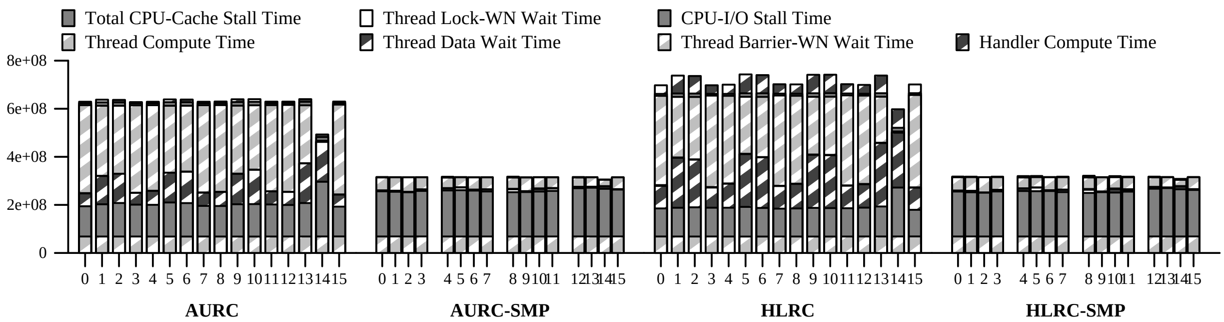

Figure 3.11: Cost breakdown for Ocean-contiguous (514x514) for AURC and HLRC.

Cost Breakdown	FFT		LU-cont		Ocean-cont		Water-nsq	
Protocol	9%	-2%	1%	0%	12%	-11%	5%	-3%
I/O Wait	12%	-8%	2%	-1%	16%	-16%	6%	-4%
Barriers	29%	2%	91%	-33%	441%	-373%	24%	-3%
Locks	0%	0%	0%	0%	1%	1%	41%	1%
Data Wait	109%	-26%	17%	-6%	139%	-131%	16%	-11%
CPU Stall	89%	34%	42%	3%	203%	75%	15%	0%
Compute	100%	0%	114%	0%	100%	0%	104%	0%
Barrier Wait	24%	4%	60%	-5%	344%	-304%	18%	2%
Lock Wait	0%	0%	0%	0%	0%	1%	17%	11%
PFetch time	21625	28443	20695	9417	20069	20156	23765	11311
# PFetches	2705	-1793	258	-135	4591	-4434	394	-298
# Local Locks	0	0	0	0	0	5	0	748
# Remote Locks	0	0	0	0	15	-6	1170	-748

Table 3.5: Changes in Protocol Costs from AURC to AURC-SMP for the regular applications.

SMP node are adjacent in the grid then much of the sharing will be contained locally. The number of page fetches is greatly reduced, and performance improves even though the cost per–page fetch goes up due to contention at the bus and the node–to–network interfaces. In Ocean-contiguous the data accessed by each processor are mostly allocated in the local SMP node. The lower data wait time results in lower synchronization time.

We see that AURC improves significantly by using SMPs. If we look at the efficiency factor of AURC for Ocean-contiguous, we see that it is very low for the uniprocessor node case, and much better for the SMP case. This is because the relatively good speedup in Ocean-contiguous in the uniprocessor case comes mainly from cache effects. The application, in terms of the protocol overheads, performs very poorly, which gives a low efficiency factor. In the SMP case, the protocol overheads reduce dramatically, and this is captured by the efficiency factors.

AURC-SMP gives a speedup of around 16. As was mentioned before, the super–

Cost Breakdown	Barnes-rebuild		Radix		Volrend		Water-spatial	
Protocol	8%	-3%	4%	-2%	7%	-4%	2%	-1%
I/O Wait	6%	-2%	6%	-4%	6%	-4%	2%	-2%
Barriers	28%	-11%	171%	296%	2%	-1%	67%	-10%
Locks	162%	-26%	20%	243%	61%	-35%	15%	3%
Data Wait	32%	-11%	336%	279%	36%	-25%	29%	-9%
CPU Stall	12%	0%	10%	1%	17%	-1%	17%	-1%
Compute	100%	0%	100%	0%	133%	-8%	103%	0%
Barrier Wait	26%	-10%	164%	282%	1%	-1%	57%	-9%
Lock Wait	150%	-24%	14%	75%	53%	-31%	14%	3%
PFetch time	20850	8232	172213	1140274	24025	4480	28062	59742
# PFetches	3243	-1740	2082	-1565	1136	-808	511	-399
# Local Locks	1	511	1	7	0	223	0	6
# Remote Locks	2226	-514	47	-7	430	-200	19	-5

Table 3.6: Changes in Protocol Costs from AURC to AURC-SMP for the irregular applications.

linear speedup is due to cache effects. More precisely, the sequential run suffers from very high local stall time (2-3 times the compute time). The parallel application takes advantage of the smaller working set size in each processor, and the stall time is reduced substantially.

Cache effects are noticeable in the parallel execution for the larger problem size as well. The working set does not fit in the cache of each processor, and the stall time is increased substantially. Since multiple processors share the same memory bus, there is a great deal of contention in the system. The application spends most of the time in waiting for local memory operations, with a CPU stall time of (268%, 261%, 296%). Thus, speedups are much lower for the SMP node case. Protocol efficiencies increase however. The main reason is that relative protocol overheads are reduced in the larger problem size and poorer performance comes from cache effects.

Cost Breakdown	FFT		LU-cont		Ocean-cont		Water-nsq	
Protocol	57%	-31%	11%	-6%	82%	-79%	5%	0%
I/O Wait	11%	-7%	2%	-1%	16%	-16%	8%	-4%
Barriers	42%	-10%	100%	-39%	426%	-355%	23%	3%
Locks	0%	0%	1%	-1%	2%	0%	57%	-2%
Data Wait	153%	-49%	29%	-14%	231%	-222%	13%	-7%
CPU Stall	63%	39%	37%	4%	181%	94%	15%	0%
Compute	100%	0%	114%	0%	100%	0%	104%	0%
Barrier Wait	28%	1%	70%	-12%	364%	-319%	20%	5%
Lock Wait	0%	0%	1%	-1%	1%	1%	18%	7%
PFetch time	32395	27367	31978	6890	33270	10448	19687	18655
# PFetches	2541	-1569	285	-162	4647	-4492	382	-282
# Local Locks	0	0	0	0	0	5	0	725
# Remote Locks	0	0	0	0	15	-6	1170	-725

Table 3.7: Changes in Protocol Costs from HLRC to HLRC-SMP for the regular applications.

Second group

In the second group are LU and Barnes-rebuild (Figure 3.12), where the improvement comes again from sharing of data and lower synchronization costs as well, but at a much smaller degree than Ocean. These are applications for which clustering helps in data sharing and prefetching, but less dramatically than in in Ocean. Cheaper synchronization also makes a noticeable difference.

LU: The improvement in performance compared to the uniprocessor configuration comes from cheaper synchronization, namely hierarchical barriers, and sharing of data in each node, which results in lower data wait times. For instance, the barrier cost is reduced by 33% in AURC-SMP and by 39% in HLRC-SMP. Similarly, the reduction in data wait time is 6% and 14% respectively.

Cost Breakdown	Barnes-rebuild		Radix		Volrend		Water-spatial	
Protocol	8%	4%	5%	2%	7%	2%	13%	-9%
I/O Wait	8%	-3%	1233%	-1231%	9%	-4%	3%	-3%
Barriers	235%	-169%	2142%	-1849%	2%	-1%	66%	-32%
Locks	337%	-76%	9%	-7%	118%	-15%	30%	-9%
Data Wait	50%	-21%	588%	-469%	30%	-12%	36%	-26%
CPU Stall	13%	-1%	10%	1%	18%	1%	10%	3%
Compute	100%	0%	100%	0%	143%	6%	103%	0%
Barrier Wait	181%	-116%	2012%	-1741%	1%	0%	57%	-25%
Lock Wait	289%	-62%	2%	-1%	79%	-4%	26%	-7%
PFetch time	28459	11698	298878	-38977	20077	17310	34089	5742
# PFetches	3298	-1778	2056	-1584	1033	-699	543	-408
# Local Locks	1	528	1	6	0	192	0	7
# Remote Locks	2228	-531	47	-6	438	-188	17	-4

Table 3.8: Changes in Protocol Costs from HLRC to HLRC-SMP for the irregular applications.

Barnes-rebuild (Figure 3.12): Barnes-rebuild performs quite poorly under both protocols due to extremely high lock costs. In the SMP case improvements in data wait time due to sharing and prefetching, and in lock times due to local acquires, are relatively small. Overall performance improves, but not by much. In HLRC-SMP nodes, data wait time is smaller and more balanced, but lock acquire costs, which dominate performance, remain expensive (163%, 261%, 312%), and imbalanced. The

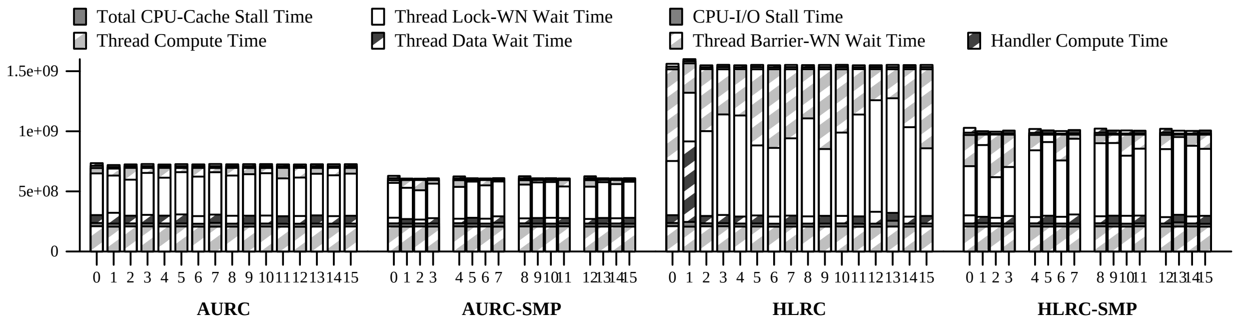

Figure 3.12: Cost breakdown for Barnes-rebuild for AURC and HLRC.

reduction in the number of remotely acquired locks due to the use of SMP nodes is not very large.

Third group

The third group in this class contains applications where there is an improvement but of a different degree for AURC-SMP and HLRC-SMP. These are Volrend (Figure 3.13), and Water-spatial (Figure 3.14).

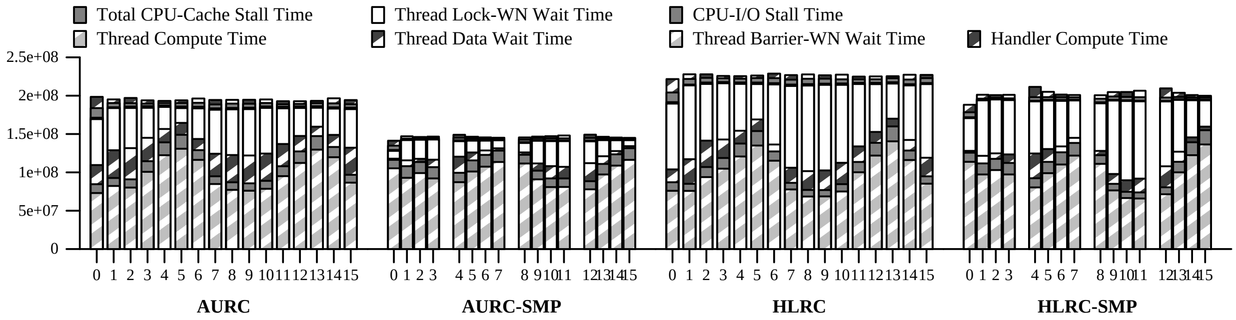

Figure 3.13: Cost breakdown for Volrend for AURC and HLRC.

Volrend (Figure 3.13): Volrend uses a task stealing mechanism to achieve load balancing. This mechanism uses locks to perform atomic operations on the task queue. Using SMPs results in an improvement in all protocol costs without introducing additional problems. In AURC-SMP page fetches are reduced by about 60% due to sharing and prefetching. Moreover, lock time is reduced by about 50% and computation is not as imbalanced, since the task stealing method takes advantage of the multiple nodes per SMP (it tries to steal from local processors first, converting remote locks and communication to inexpensive local locks and communication). HLRC-SMP improves somewhat but not as much as AURC-SMP. The main reason is that lock imbalances are still present because remote locks are more expensive in

HLRC due to the cost of diffs, and the distribution of local versus remote locks in each processor is somewhat more uneven in HLRC, which leads to higher imbalances.

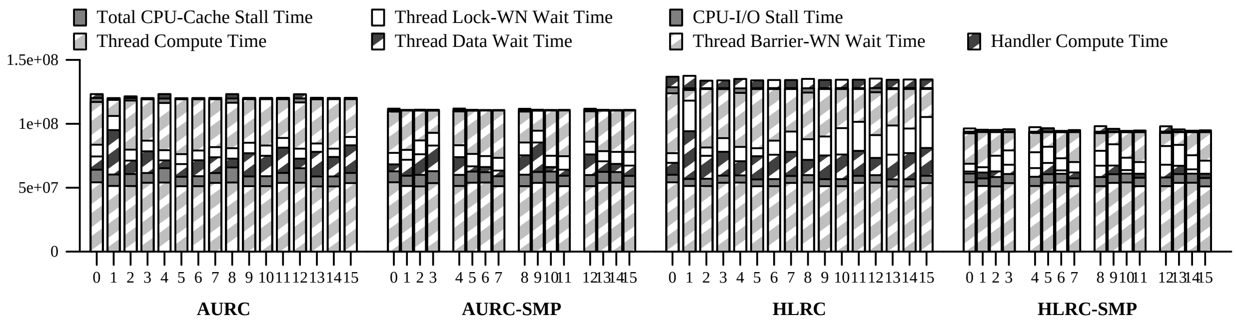

Figure 3.14: Cost breakdown for Water-spatial for AURC and HLRC.

Water-spatial (Figure 3.14): In AURC-SMP performance improves slightly compared to AURC. Total data wait time is reduced, but it is not balanced among processors. The simulator shows that the reason for the imbalance is contention in the outgoing queue in some network interfaces because of automatic update traffic, which slows the progress of outgoing requests. The net improvement in performance is small.

In HLRC-SMP the picture changes. HLRC-SMP performs considerably better than HLRC. Data wait time decreases considerably because of sharing and prefetching. Also synchronization costs are much lower. Unlike AURC-SMP, though, protocol overheads remain balanced and the improvement in performance is larger. There is practically no contention in the outgoing queue, and request messages are sent out immediately for all the processors.

3.7.2 Second class

The second class of applications consists of FFT (Figure 3.15), Barnes-spatial (Figure 3.16), and Water-nsquared. These applications do not benefit (or benefit little) from the use of SMP nodes with either protocol.

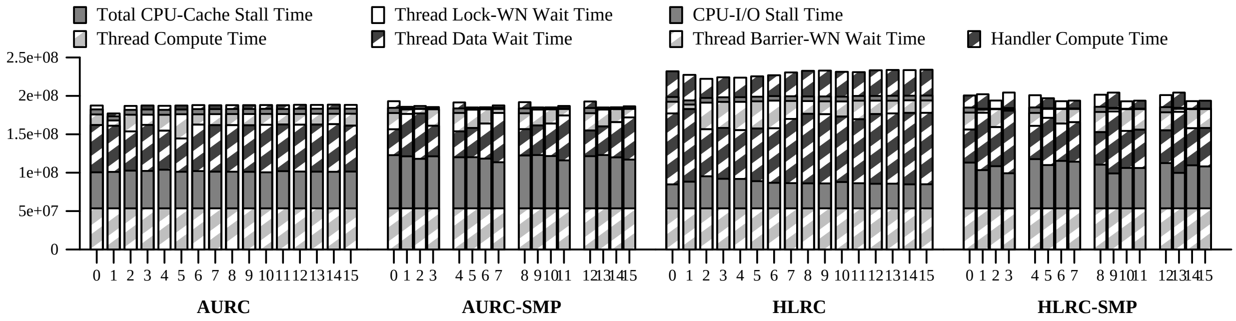

Figure 3.15: Cost breakdown for the 1M FFT for AURC and HLRC.

FFT (Figure 3.15): Since communication is all–to–all rather than localized in the matrix transpose in FFT, the clustering accomplished by using SMP nodes reduces the amount of inherent remote communication by roughly a factor of k/p, where k is the number of processors in an SMP node, and n is the total number of processors. However, in the SMP configuration we find that page fetches are reduced dramatically for the small problem size compared to the uniprocessor node case. This is because the problem size is such that the data that need to be fetched by different processors lie on the same page. Using multiple processors in each node thus results in substantial prefetching and in a reduction in data wait time. However, data wait time is imbalanced among processors within a node because the number of page fetches differs among them, so the overall reduction does not translate to a large improvement in performance. Barrier synchronization time is high due to this imbalance, and protocol efficiency low.

For the larger problem size, speedups and efficiency factors are somewhat better in all cases because less of the data that are fetched on a page are wasted. In both AURC-SMP and HLRC-SMP data wait time once again improves because of sharing and prefetching, but imbalances in the data wait time limit the effect on performance. Moreover, with the larger working sets, local memory stall time is considerably higher than in the uniprocessor node configuration due to contention on the memory bus. For instance, in AURC-SMP local stall time is (112%, 115%, 129%) as opposed to (87%, 84%, 93%) in AURC. (i.e., 112% minimum across processors, 115% average, and 129% maximum, and so on).

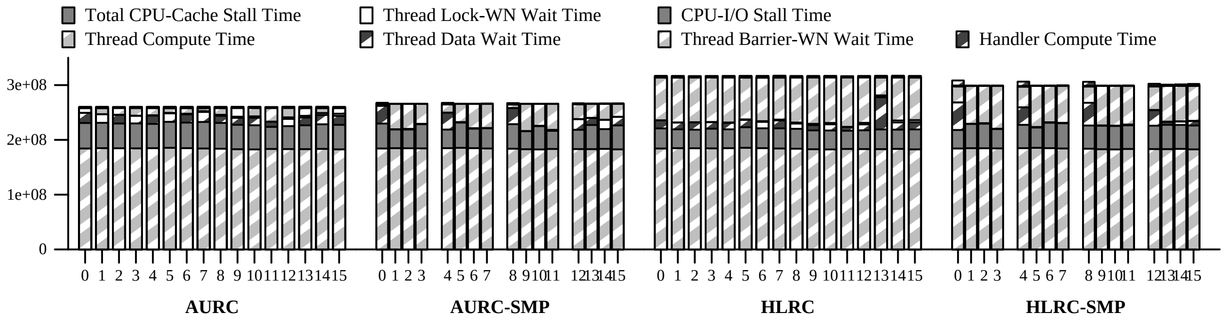

Figure 3.16: Cost breakdown for Barnes-spatial for AURC and HLRC.

Barnes-spatial (Figure 3.16): As in the uniprocessor configuration, Barnes-spatial performs very well under all protocols in the SMP case as well. Clustering does not help much. The reason is that while the barriers that this application uses often in tree building (instead of locks) are cheaper, data wait time is still imbalanced due to different number of page fetches among processors. This shows up as synchronization wait time, and the benefits from clustering are negligible.

Water-nsquared: Here there is a large overall reduction in inter–node communication in the SMP configuration due to prefetching, but as in FFT imbalances are created in the numbers of page fetches among processors within an SMP. The same is true for lock accesses. Thus, despite the large overall reduction in communication, the improvement in performance is small.

In the applications in this second class, using SMP nodes does reduce aggregate communication and synchronization costs substantially due to sharing and prefetching, but the increases often do not translate to large performance increases because the reductions are imbalanced on a per–processor basis within the nodes.

3.7.3 Third class

In the third class of applications are Radix and Raytrace (Figure 3.17). These exhibit different relative behavior under different protocols.

Radix: All protocols perform very poorly with Radix. Among the AURC protocols, AURC-SMP performs worse than AURC. The amount of data sent as replies to page requests decreases to about one fourth in the SMP configuration, as with FFT, yet performance is much worse because AU traffic per processor is about the same as in AURC, while bus and network interface bandwidth are now shared. This creates contention in all components of the path between the sender and the receiver, delaying not only AU messages but also and request/control messages that get stuck behind them, as can be seen from the simulator. Consequently, all forms of communication and synchronization are slowed down. For instance the average page fetch cost is huge, (466935, 1312487, 1942957) cycles. Protocol efficiency is very low for AURC-SMP.

HLRC-SMP performs much better than HLRC. HLRC-SMP does not suffer from increased traffic as much as AURC-SMP, since there is no automatic update traffic. Messages are delivered faster, and fetch time, lock time and barrier costs are much smaller, so performance improves. However, SVM protocols cannot handle the bursty, scattered, remote–write communication of Radix, and do not do well overall.

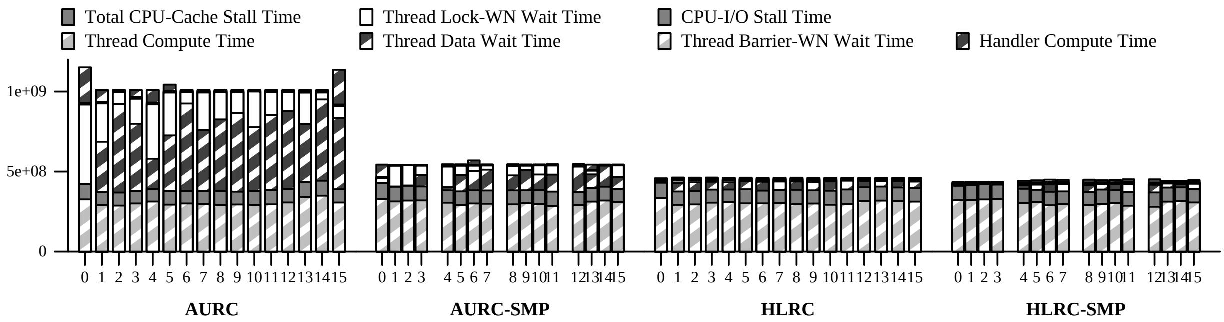

Figure 3.17: Cost breakdown for Raytrace for AURC and HLRC.

Raytrace (Figure 3.17): In Raytrace, AURC benefits greatly from sharing, whereas the improvement for HLRC is very small. The reason for the improvement in AURC is the small automatic update messages that are used to update data. When SMP nodes are used, the number of messages is reduced substantially (the amount of data is reduced by a factor of 4 with 4–way SMPs), and as a result the contention problem is much less severe than the uniprocessor node case. In HLRC the number of messages is not a problem since coalesced, large messages (diffs) are used to update the home nodes. Thus the performance of HLRC is much better than AURC and the benefit of using SMP nodes is much smaller.

3.8 Hardware Support for Automatic Update on SMPs

As mentioned above, some recent network interfaces also provide hardware support for the propagation of writes at fine granularity (a word or a cache line, say) to a remotely mapped page of memory [16, 50, 37]. This facility can be used to accelerate home–based protocols by eliminating the need for diffs, leading to a protocol called automatic update release consistency or AURC [46]. AURC [44] is the version of

HLRC that utilizes hardware support for automatic update in the SHRIMP multicomputer to alleviate certain protocol bottlenecks. The extensions described above for HLRC apply also to AURC and result in AURC-SMP. However, AURC-SMP suffers in several aspects, mainly because it requires write–through caching. Each write to local memory needs to be seen from the network interface, and thus all local writes to mapped regions of memory have to appear on the memory bus. This leads to the requirement for write–through caching for mapped regions, that increases the traffic in the memory bus and the network.

Write–back AURC-SMP (WBAURC-SMP) is a new protocol that provides AU support with write–back caches. WBAURC-SMP addresses two problems with AURC-SMP: (i) AURC-SMP may not be even implementable on newer systems which do not support write–through caches. (ii) Since AURC-SMP maps shared pages write–through, it can unduly stress the memory and I/O buses, especially with SMP nodes [1].

The problem with write–back caches for AU support is that not all writes are visible to the memory bus snooper to be propagated to the home. In WBAURC-SMP changes are propagated to remote homes either when modified lines are replaced or by flushing modified cache lines at release time [2]. In addition to snooping hardware, extra hardware support is needed to identify which words were modified in each cache line and only propagate those words. Otherwise, if entire cache lines are propagated, valid words at the home from more recent writes to other parts of the cache line could be overwritten by older values due to false sharing. One possible implementation, which we assume, is to add a dirty bit *per word* to the second level cache. Caches can either flush only the dirty words, or an external agent on the bus can snoop the evicted

[1] Pages in the home nodes can either be mapped write–through, as assumed in [44], or can be mapped write–back since coherent DMA will provide the latest data to an incoming page fetch request; we assume the latter here since it performs better especially with SMPs.

[2] If the processor architecture does not allow flushes at user level, this may have to be done through a system call.

cache lines and propagate only the dirty words. Other possibilities include having the memory controller perform diffs in hardware on cache lines as they are written back to local memory, to detect the modified words. The basic issues in all these cases is the amount of traffic that appears on the memory bus and the implementation complexity and cost.

This section presents our results for AU support. We present results for AURC-SMP, HLRC-SMP and WBAURC-SMP on a system with four, four–way SMP nodes for a total of 16 processors. All protocols use time–stamps to maintain ordering of events and they are home–based. The key difference between them is the method used for propagating modified data to the home. As mentioned before, AURC-SMP uses an AU mechanism, HLRC-SMP is the all–software version and WBAURC-SMP is the improved AURC-SMP version, where caches do not need to be configure as write–through.

We first assume that a commodity NI like Myrinet (with AU support) is used. To address the question of whether AU is valuable only with customized NIs, we also discuss how the results change when an NI, customized for efficient AU propagation is used. We present overall performance data in two main forms: tables with speedups and protocol efficiencies, and per–processor bar–graphs with execution time breakdowns for the most interesting cases (Figures 3.18–3.22).

For the *regular* applications, FFT, LU, and Ocean, all three protocols perform very similarly for the better (contiguous) versions and very poorly for the others (Table 3.9). The contiguous versions are essentially single writer applications at page granularity as discussed earlier. With proper data placement, they do not exercise the multiple writer features that distinguish these protocols. There is essentially no update or write through traffic, and no diffs. The irregular applications are more interesting.

Application	Problem Size	Speedups AURC SMP	Speedups HLRC SMP	Speedups WBAURC SMP	Efficiency Factors (%) AURC SMP	Efficiency Factors (%) HLRC SMP	Efficiency Factors (%) WB AURC SMP
FFT	18	6.20	5.74	5.99	42	40	45
FFT	20	8.29	8.28	8.29	65	59	65
LU-contiguous	512	12.42	12.25	12.39	68	66	68
Ocean-contiguous	258	16.86	15.23	15.33	51	47	49
Ocean-contiguous	514	12.83	12.80	13.17	82	79	81
Barnes-rebuild	16K	6.24	3.82	6.08	39	24	38
Barnes-spatial	8K	12.81	12.19	13.28	83	80	84
Barnes-spatial	16K	12.95	11.52	13.56	84	76	85
Radix	1M	1.25	3.41	3.72	8	21	23
Raytrace	car	11.82	14.79	10.80	72	90	66
Volrend	head	11.95	8.81	11.86	76	56	77
Water-nsquared	512	9.71	8.84	9.73	62	57	63
Water-spatial	512	8.52	10.05	10.48	55	63	67

Table 3.9: Speedups and efficiency factors (as %) for the SMP configuration.

Barnes-rebuild (Figure 3.18): Barnes-rebuild has an efficiency of less than 40% in all three protocols due to extremely high lock wait time [3]. Most of this time is spent in waiting to acquire the locks and not in protocol processing. This effect is exaggerated due to the dilation of critical sections because of page faults occurring in them, as discussed in Section 3.6. HLRC-SMP performs worse by a factor of two compared to AURC-SMP, mainly because of the much higher lock wait time: diff generation and application happen at synchronization time as well, increasing serialization at locks. WBAURC-SMP performs similarly to AURC-SMP, since traffic is not a major issue.

Barnes-spatial (Figure 3.19): By eliminating the locks, Barnes-spatial performs very well under all protocols (Table 3.9). The main overhead is barrier cost. HLRC-

[3]Locks are introduced to label point to point flag synchronization. When they are replaced by separate *acquire* and *release* primitives, the wait time is spent at those instead

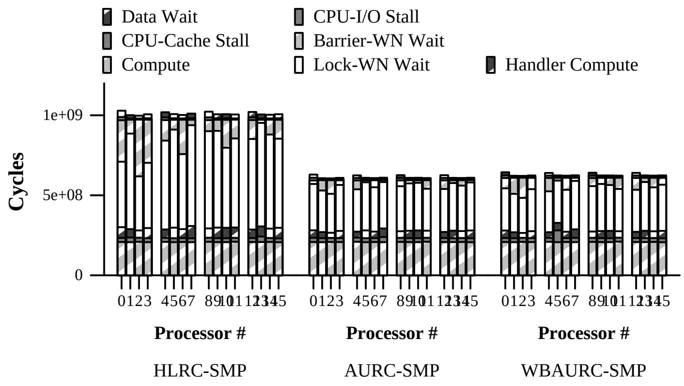

Figure 3.18: Cost breakdown for Barnes-rebuild for HLRC-SMP, AURC-SMP and WBAURC-SMP.

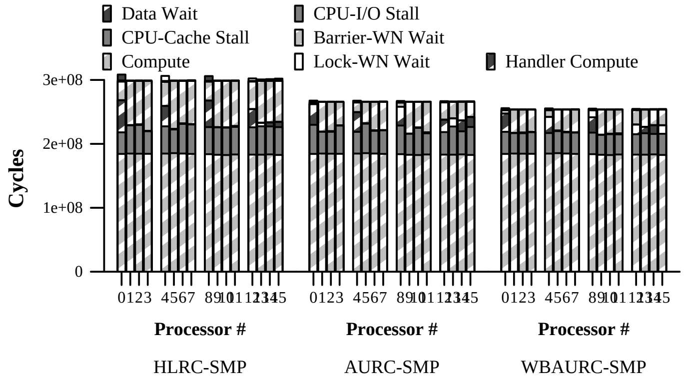

Figure 3.19: Cost breakdown for Barnes-spatial for HLRC-SMP, AURC-SMP and WBAURC-SMP.

SMP is somewhat worse because of the increased barrier cost due to diff computation and application. Restructuring the application not only increases performance greatly under all protocols, but also diminishes the differences among them.

Radix: AURC-SMP performs very poorly with Radix with a speedup of 1.25. The reason for the very low performance is high data wait and synchronization times. Contention created by high AU traffic due to remote writes results in high page fault costs. The number of page faults is similar across processors but the costs vary significantly (from about 450K to 1900K cycles per–page fetch) and are higher than the uncontended page fetch cost due to contention (the uncontended page fetch cost is about 15K cycles). Moreover, the simulator shows that control messages in the outgoing and incoming network queues are delayed for the same reason. Most of the synchronization time is wait time due to imbalances and not the protocol overhead itself, although the imbalances are not in computation but are caused by contention. A lot of traffic is incurred just before barriers, further increasing their cost.

HLRC-SMP performs better than AURC-SMP (speedup is 3.41) because of the smaller number of messages. Updates to shared data in HLRC-SMP occur not at every write but at a release, which results in fewer and larger messages. The messages are delivered faster due to less contention, so fetch time, lock time and barrier costs are much smaller and performance improves.

WBAURC-SMP does not have the increased traffic problems of AURC-SMP and thus regains the loss in speedup, improving it from 1.25 to 3.72. Data wait times and barrier costs are reduced. However, imbalances are still present in data wait times.

In general, scattered remote writes and high communication make Radix a difficult application for SVM systems, particularly for AURC-SMP due to increased write–through AU traffic. As in other applications, WBAURC-SMP is the best performing protocol. To reduce scattering of writes and hence traffic and contention, we tried a version of Radix that first gathers the changes locally in a copy buffer and then propagates them to the remote nodes. This improved performance on 16 processors by a factor of 1.5-2 across protocols.

Raytrace (Figure 3.20): All protocols perform very well for this application. HLRC-SMP performs better than AURC-SMP and WBAURC-SMP because of the smaller and better balanced data wait time. Although the total traffic is similar in all protocols, HLRC-SMP uses fewer, bigger messages for the update mechanism and the simulator shows that this reduces contention in the network queues. This is the

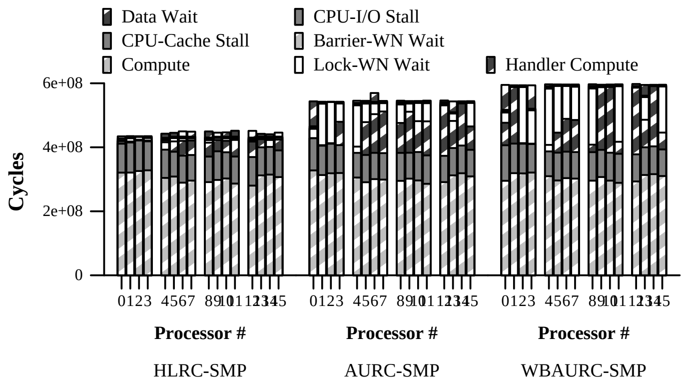

Figure 3.20: Cost breakdown for Raytrace for HLRC-SMP, AURC-SMP and WBAURC-SMP.

only application where WBAURC-SMP performs worse than HLRC-SMP.

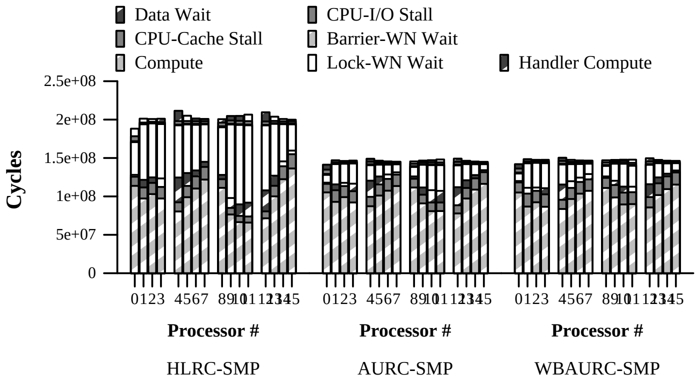

Figure 3.21: Cost breakdown for Volrend for HLRC-SMP, AURC-SMP and WBAURC-SMP.

Volrend (Figure 3.21): Data traffic and wait time are not very substantial in Volrend. AURC-SMP does very well. The only problem is created from imbalances in the compute time and high lock wait time due to the high overhead of task stealing [47]. Locks in Volrend protect task queue entries and hence migratory data. For all protocols, many processors contend for the same, relatively small number of locks, and requests for locks are often stuck waiting in queues for other messages to be delivered first or for protocol handlers to run. First, contention occurs at the homes of the locks, especially since they are not assigned in a clever way for locality. Second, forwarded lock requests are queued at the owner, waiting for earlier messages or for the current lock to actually arrive there. Third, page faults that occur within critical sections often have to be satisfied remotely (especially when the locks and protected data are migratory), dilating the critical section [46, 47]. Observed lock wait times are at least an order of magnitude longer than the uncontended time for accessing an owned lock, and the times are imbalanced due to the different numbers of local and remote locks acquired by each processor. These times are much larger than in Barnes, where the problem is not so much the contention at locks as the sheer number of locks executed. In HLRC-SMP in particular, the migratory locks in Volrend also causes a lot of diffing and diff transfers, further slowing down acquires as well as releases. This last increase in lock time and serialization is alleviated with AU-based methods, which do not require diffs; together with the lack of need for interrupts on write propagations, this leads to much better performance. The behavior of WBAURC-SMP is similar to AURC-SMP.

Water-nsquared: All protocols perform well with Water-nsquared since traffic is low and much of the application is single-writer. Protocol overheads are low and balanced once again. HLRC-SMP is somewhat worse than the other two protocols due to more expensive locks. Improving locks would be the best way to improve Water-nsquared performance.

Water-spatial (Figure 3.22): AURC-SMP performs at about 50% parallel efficiency with Water-spatial. However, AU traffic creates contention in the outgoing network queues. The simulator shows that this slows down outgoing page requests, which get stuck behind AU traffic, and results in imbalances in data wait times. In

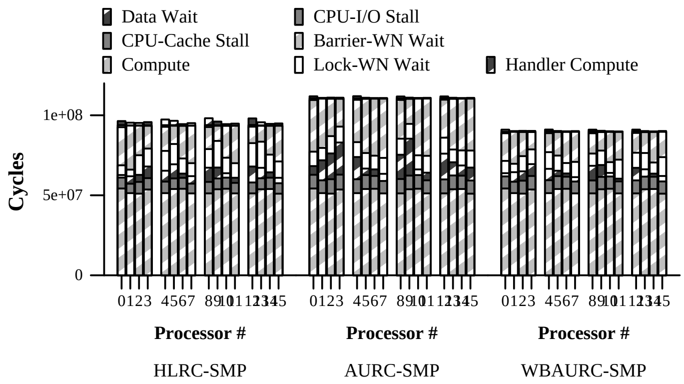

Figure 3.22: Cost breakdown for Water-spatial for HLRC-SMP, AURC-SMP and WBAURC-SMP.

HLRC-SMP, unlike AURC-SMP, protocol overheads are balanced and performance is considerably higher. There is no contention in the outgoing network queues and request messages are sent out immediately for all the processors. Data wait time is considerably smaller, (min 3%, avg 10%, max 19%) in HLRC-SMP as opposed to (min 6%, avg 20%, max 45%) for AURC-SMP. WBAURC-SMP performs best by a significant amount. It does not suffer from AURC-SMP's contention–induced load balancing problems.

Summary: For regular applications, hardware support for AU is not particularly useful. The nature of these applications is such that structuring them well causes the additional hardware support to not be used, and not structuring them well will result in poor performance regardless of the protocol.

With irregular applications AURC-SMP helps substantially over HLRC-SMP in cases where the lock cost is the main problem (Barnes-rebuild, Volrend and Water-nsquared). Since lock cost is higher for HLRC-SMP due to diffs, serialization effects in lock acquisition result in worse performance (e.g., when task stealing or migratory patterns are used). On the other hand AURC-SMP does worse than the all-software HLRC-SMP for applications where AU or other traffic is the problem (Radix, Raytrace and Water-spatial). In these cases, imbalances in communication and synchronization times created by contention are worse in AURC-SMP due to the increased traffic. Traffic is not only a problem for data transfers, but control messages (e.g., requests) get stuck behind data messages (e.g., AU) even when inherent data traffic is not very high, suggesting that separate queues for data and control messages or different priorities might be useful. Where traffic was a problem for AU we experimented with higher bandwidth buses, but these did not help very much. The real problem was that a few messages of a latency–critical type (e.g., page requests or synchronization messages) were stuck behind less critical AU packets. WBAURC-SMP is always closest to or better than the best performing of the other two protocols. In all other cases the differences among protocols are not very large, at least with these commodity network interfaces. An interesting observation is that improvements to the applications usually tend to increase performance much more than automatic update support, and these improvements tend to reduce the differences in protocol performance.

We also compared the protocols under consideration in a system with uniprocessor nodes. We find that the conclusions from the SMP node configuration hold in this case as well.

3.8.1 Customized network interfaces

We also examined the impact of customized network interfaces like the one in SHRIMP [16] on AU based protocols. A customized NI allows for much shorter packet preparation overheads since it consists of a dedicated state machine instead of a more general purpose processor on commodity oriented interfaces. Customized NIs incur lower occupancy and they result in higher benefits for protocols that make use of automatic update support, which tend to transfer more packets. The occupancy for customized NIs is set to about 100 cycles as opposed to about 1000 for the commodity NIs.

The results show that the performance of AURC-SMP improves for the two applications, Radix and Water-spatial, where raw AU traffic creates contention in the network queues. The speedup for Radix is improved by a factor of more than 2, since contention between AU and page traffic is reduced, and the effect of contention between AU and requests in Water-spatial disappears. For Radix, WBAURC-SMP improves for the same cases and the result is a speedup of more than 9 for WBAURC-SMP. Also the speedup of Raytrace improves to almost 13.5 in both AURC-SMP and WBAURC-SMP. The performance improvement in the other applications, which do not suffer from very high AU traffic, is marginal. Customized NIs can thus make AU-based SVM more attractive for some applications.

3.9 Related Work

The major advantages and disadvantages of a shared address space programming abstraction compared to explicit message passing are described in [21] (Chapter 3) and [82] and are not covered here. Several papers have discussed the design and performance of shared virtual memory for SMPs [19, 49, 29, 76, 91] for different protocols. We focus here mostly on studies that propose or evaluate the use of SMP nodes and hardware support.

A messaging passing programming abstraction extended to support the two level hierarchy in a network of SMPs is presented in [61], where the authors find that achieving performance benefits on this architecture is not straight–forward; the mismatch between the native shared memory and the imposed message passing abstractions within the SMP nodes entails many issues that require careful consideration.

Moreover, to achieve good performance compared to a network of uniprocessors, application issues like data layout have to be dealt with in ways similar to SVM systems.

AURC and LRC have been compared in [44, 46] and HLRC and LRC have been compared for some applications on the Intel Paragon in [100]. SVM on SMPs is argued to be a promising direction in [19]. In [49] the authors find that a reserved protocol processor in not required. Earlier versions of this work through simulations can be found in [8, 9].

The SoftFLASH system [29] has goals similar to ours, but uses a single–writer protocol (modeled after the FLASH protocol). The protocol is not as aggressive as other multiple–writer lazy SVM protocols. The system exhibits very high latency and expensive TLB synchronization. The use of 16 KBytes pages in SoftFLASH increases false–sharing and causes fragmentation that result in greater bandwidth needs. We use a multiple–writer home–based protocol based on lazy release consistency (LRC). The network interface we use provides lower latency and higher bandwidth. We use 4 KBytes pages and the safe page fetch helps us avoid the TLB synchronization problem.

The Multigrain Shared Memory System (MGS) [99] built on top of the MIT Alewife machine uses a protocol very similar to the SoftFLASH system. The system is implemented by partitioning the Alewife machine, so each node is a distributed rather than centralized shared memory multiprocessor, and the number of external network connections from the node (and hence bandwidth) scales linearly with the number of processors in it. The processors in Alewife are very slow.

The Cashmere–2L system [91] at Rochester is the closest to our system. It uses a multi–writer, directory–based scheme that is an eager variant of release consistency. This is unlike our lazy, vector time–stamp based scheme. The platform is a cluster of Alpha SMPs connected by the Memory Channel network. The protocol takes great advantage of particular features found on the Memory Channel network interface that are not found in other interfaces (such as Myrinet).

The Shasta system [77] performs the coherence in software at cache–line granularity by instrumenting the executable to insert access control operations before shared loads and stores. This idea was first implemented in the Blizzard–S system [79]. There is currently an implementation of Shasta that runs on a cluster of Alpha SMP nodes [76]. This implementation shows promising performance. However, it is not page–based shared virtual memory. It has control–data structure requirements that are dependent on the number of processors which may lead to scalability problems for large number of processors. Also, it relies on code instrumentation to detect accesses to share data.

The problem of write through caches is mentioned but not dealt with in an earlier work [55], in the context of different SVM protocols and uniprocessor systems. [8] is a preliminary version of this work. For this work, we use a much more detailed simulation environment and improved implementations of the protocols.

Erlichson et al. [29] conclude that the transition from uniprocessor to multiprocessor nodes is nontrivial, and performance can be seriously affected unless great care is taken. The protocol and system they assume is very different and less aggressive than ours.

Bianchini et al. in [7] propose diffing in hardware as another form of hardware support for SVM. They assume write through caches and uniprocessor nodes. A dedicated protocol processor performs diffs on–the–fly and offloads overheads from the computation processor. Their simulations conclude that using the coprocessor can double the performance of the TreadMarks LRC system [51] on a 16–node configuration. Results for using a coprocessor for diff computation and application for HLRC on the Intel Paragon are far less optimistic [100].

Holt et al. present a metric similar to protocol efficiency [41]. They use it to characterize the performance of applications on a hardware cache coherent machine. Our results for AURC on uniprocessor systems are consistent with the results obtained in [46].

3.10 Discussion and Conclusions

This chapter has described an SVM system for SMP nodes, which attempts to use the hardware sharing within the SMP as much as possible and reduce the frequency of software protocol involvement, and we have tried to understand its performance, particularly in comparison with the baseline SVM protocol across uniprocessors.

We find that for the same total processor count, using SMP nodes improves performance for both the all–software HLRC protocol and the AURC protocol that uses hardware remote write support, particularly when the protocols and applications are designed and used properly. This is despite the fact that the traffic pressure on the memory and I/O buses tends to increase. In some cases imbalances are introduced because of increased contention, and overall benefits are reduced. However, the reduction in the number of remote page fetches and in synchronization cost tends to overshadow the increased contention, resulting in a small improvement in some applications and a substantial improvement in others. Preliminary experiments show that even when performance degrades due to contention on the I/O bus (which interfaces to the network), it can be alleviated by increasing I/O bus bandwidth with the number of processors in an SMP node.

Out of ten applications, both protocols improve substantially with the use of SMPs in five of them, in three there is a smaller improvement (or they perform the same as in the uniprocessor node case), and for the other two results differ for each protocol with AURC-SMP performing worse than AURC for one application. For two out of the three regular applications performance improves significantly with SMP nodes for both protocols. An exception to this is FFT, with its all–to–all communication, which exhibits no significant improvement. Among the irregular applications, Barnes-rebuild, Volrend, and Water-spatial benefit in both protocols, Barnes-spatial and Water-nsquared do not exhibit any significant improvement, and the last two applications, Radix and Raytrace, behave differently under each protocol. Radix improves with HLRC-SMP but performs worse with AURC-SMP, and Raytrace improves with AURC-SMP and exhibits no improvement with HLRC-SMP. In many cases there is a large reduction in the number of remote (cross–node) page fetches due to the use of SMP nodes, even larger than expected from inherent interprocess communication patterns, due to the interactions with page granularity. However, performance does not increase as much as expected even in these cases, because the reduction is uneven across processors so there is significant load imbalance in communication costs.

We have also examined the effectiveness of automatic update support for shared virtual memory on modern systems. We first discussed how the original AURC protocol, which relies on write–through second–level caches, can be extended to work with modern systems that have write–back caches (WBAURC-SMP). Then we compared the performance of the three protocols, HLRC-SMP, AURC-SMP and WBAURC-SMP, on a system with commodity network interfaces. WBAURC-SMP solves the performance problems that arise with AURC-SMP due to the increased traffic in SMP systems and yields better performance than both AURC-SMP and an all–software HLRC-SMP for a range of irregular applications.

For many regular applications, hardware support for AU is not particularly useful. The nature of these applications is such that if they are structured well there is little need for diffing or write propagation, and if not then they perform poorly with all protocols.

The main advantage of AURC-SMP and WBAURC-SMP over HLRC-SMP is that, the hardware assisted protocols avoid diff computation and application. This protocol processing overhead is especially damaging to HLRC-SMP because it is incurred at synchronization points, where it can have a cascading effect on serialization. This shows up substantially in applications with either a lot of locking (e.g., Barnes-rebuild) or with a few locks that are heavily contended (e.g., Volrend due to task queues and Water-nsquared in the force updates). In the latter, lock requests have to wait a while in incoming queues while previous lock or other requests are being serviced (whether locally at the previous owner or remotely). AURC-SMP and WBAURC-SMP are substantially advantageous in these cases even with commodity NIs.

On the other hand, AURC-SMP can lead to a lot of AU traffic when the homes of pages being written are not local. It performs worse than HLRC-SMP in applications that cause a lot of raw AU and data traffic (e.g., Radix) and in which the AU traffic in queues slows down important request or control messages that are in the critical path. Separate queues and/or priorities for different types of messages might be useful here. AURC-SMP is also less robust in performance to poor placement of pages across memories. A potential disadvantage of WBAURC-SMP is that it occupies system resources with AU traffic over a larger amount of time. Overall, WBAURC-SMP is always close to the best performing protocol (with the exception of Raytrace) since it combines the advantages of both.

The use of lower–occupancy, customized rather than commodity NIs improves AURC-SMP and WBAURC-SMP performance substantially for the latter class of applications in which outgoing AU traffic is a problem; namely, Radix and to some extent Water-spatial. Barnes and Volrend for instance are hardly affected, since the problem there is lock cost exacerbated by page misses within critical sections, not raw traffic. Applications that suffer due to control messages getting stuck in queues behind outgoing or incoming AU packets are also not helped as much.

Our analysis identifies several key outstanding problems for home–based SVM systems, whether or not AU–based approaches are used; i.e. even successful use of AU does not alleviate them fully. Several of these have been observed in earlier research. In the communication layer, critical control messages like requests get stuck behind other messages in queues. Interrupts are expensive and can increase this queuing effect [12]. Possible solutions to these problems are to use priorities or separate queues for different message types, and to use a smart network interface to perform some of the data movement and synchronization functions instead of interrupting the processor [11]. In the protocol layer, the homes of locks and pages are often not assigned well in irregular applications, which can hurt performance quite a bit due to serialization and request queuing in some cases (e.g., Volrend). And highly contended locks that protect migratory data pose a problem, which can perhaps be solved by using different protocols when this pattern is detected. Protocol overheads and contention due to diffing at barriers also requires addressing. Finally, the application layer can have dramatic performance effects that exceed those of all other layers, as seen in Barnes in this chapter and more generally in an earlier study [47].

Chapter 4

SVM perfromance Bottlenecks

We saw in Chapter 3 that using SMP nodes improves the performance of SVM. However, it is not clear how much better one can do by improving the either the SVM protocols or the communication architecture. In this and the next chapters we will address these issuses.

This Chapter investigates the importance of communication architecture parameters on application performance and reveals the most important system bottlenecks with respect to the communication architecture. There has been a lot of effort in providing cost–effective SVM systems by employing software only solutions on clusters of high–end workstations coupled with high–bandwidth, low–latency commodity networks. Much of the work so far has focused on improving protocols, and there has been some work on restructuring applications to perform better on SVM systems. The result of this progress has been the promise for good performance on a range of applications at least in the 16–32 processor range. New system area networks and network interfaces provide significantly lower overhead, lower latency and higher bandwidth communication in clusters, inexpensive SMPs have become common as the nodes of these clusters, and SVM protocols are now quite mature. With this progress, it is now useful to examine what are the important system bottlenecks that stand in the way of effective parallel performance; in particular, which parameters of the communication architecture are most important to improve further relative to processor speed, which ones are already adequate on modern systems for most applications, and how will this change with technology in the future. Such information can assist system designers in determining where to focus their energies in improving performance, and users in determining what system characteristics are appropriate for their applications.

We find that the most important system cost to improve is the overhead of generating and delivering interrupts. Improving network interface (and I/O bus) bandwidth relative to processor speed helps some bandwidth–bound applications, but currently available ratios of bandwidth to processor speed are already adequate for many others. Surprisingly, neither the processor overhead for handling messages nor the occupancy of the communication interface in preparing and pushing packets through the network appear to require much improvement.

4.1 Methodology

Chapter 3.5 presents the details of the architectural simulator. In this section we focus on the methodology used in this study.

As mentioned earlier, we focus on the following performance parameters of the communication architecture: host overhead, I/O bus bandwidth, network interface occupancy, and interrupt cost. We do not examine network link latency, since it is a small and usually constant part of the end–to–end latency, in system area networks (SAN). These parameters describe the basic features of the communication subsystem. The rest of the parameters in the system, for example cache and memory configuration, total number of processors, etc. remain constant.

When a message is exchanged between two hosts, it is put in a post queue at the network interface. In an asynchronous send operation, which we assume, the sender is free to continue with useful work. The network interface processes the request, prepares packets, and queues them in an outgoing network queue, incurring an occupancy per packet. After transmission, each packet enters an incoming network queue at the receiver, where it is processed by the network interface and then deposited directly in host memory without causing an interrupt [16, 24]. Thus, the interrupt cost is an overhead related not so much to data transfer but to processing requests.

While we examine a range of values for each parameter, in varying a parameter we usually keep the others fixed at the set of *achievable* values (see Section 1.3). Recall that these are the values we might consider achievable currently, on systems that provide optimized operating system support for interrupts. We choose relatively aggressive fixed values so that the effects of the parameter being varied are observed.

In more detail:

- Host Overhead is the time the host processor itself is busy sending a message. The range of this parameter is from a few cycles to post a send in systems that support asynchronous sends, up to the time needed to transfer the message data from the host memory to the network interface when synchronous sends are used. If asynchronous sends are available, an achievable value for the host overhead is a few hundred processor cycles. Recall that there is no processor overhead for a data transfer at the destination end. The range of values we consider is between 0 (or almost 0) processor cycles and 10000 processor cycles (about $50\mu s$ with a $5ns$ processor clock). Systems that support asynchronous sends will probably be closer to the smaller values and systems with synchronous sends will be closer to the higher values depending on the message size. The achievable value we use is an overhead of 600 processor cycles per message.

- The I/O Bus Bandwidth determines the host to network bandwidth (relative to processor speed). In contemporary systems this is the limiting hardware component for the available node–to–network bandwidth; network links and memory buses tend to be much faster. The range of values for the I/O bus bandwidth is from 0.25 MBytes per–processor clock MHz up to 2 MBytes per–processor clock MHz (or 50 MBytes/s to 400 MBytes/s assuming a 200 MHz processor clock). The achievable value is 0.5 MBytes/MHz, or 100 MBytes/s assuming a 200 MHz processor clock.

- Network Interface Occupancy is the time spent on the network interface preparing each packet. Network interfaces employ either custom state machines or network processors (general purpose or custom designs) to perform this processing. Thus, processing costs on the network interface vary widely. We vary the occupancy of the network interface from almost 0 to 10000 processor cycles (about $50\mu s$ with a $5ns$ processor clock) per packet. The achievable value we use is 1000 main processor cycles, or about $5\mu s$ assuming a 200 MHz processor clock. This value is realistic for the currently available programmable NIs, given that the programmable communication assist on the NI is usually much slower than the main processor.

- Interrupt cost is the cost to issue an interrupt between two processors in the same SMP node, or the cost to interrupt a processor from the network interface. It includes the cost of context switches and operating system processing. Although the interrupt cost is not a parameter of the communication subsystem, it is an important aspect of SVM systems. Interrupt cost depends on the operating system used; it can vary greatly from system to system, affecting the performance portability of SVM across different platforms. We therefore vary the interrupt cost from free interrupts (0 processor cycles) to 50000 processor cycles for both issuing and delivering an interrupt (total 100000 processor cycles or 500 μs with a 5ns processor clock). The achievable value we use is 500 processor cycles, which results in a cost of 1000 cycles for a null interrupt. This choice is significantly more aggressive than what current operating systems provide. However it is achievable with fast interrupt technology [92]. We use it as the achievable value when varying other parameters to ensure that interrupt cost does not swamp out the effects of varying those parameters.

To capture the effects of each parameter separately, we keep the other parameters fixed at their achievable values. Where necessary, we also perform additional guided simulations to further clarify the results.

In addition to the results obtained by varying parameters and the results obtained for the achievable parameter values, an interesting result is the speedup obtained by using the best value in our range for each parameter. This limits the performance that can be obtained by improving the communication architecture within our range of parameters. The parameter values for the best configuration are: host overhead is 0 processor cycles, I/O bus bandwidth is equal to the memory bus bandwidth, network interface occupancy per packet is 200 processor cycles and total interrupt cost is 0 processor cycles. In this best configuration, contention is still modeled since the values for the other system parameters are still nonzero. Table 4.1 summarizes the values of each parameter. With a 200 MHz processor, the achievable set of values discussed above assumes the parameter values: host overhead is 600 processor cycles, memory bus bandwidth is 400 MBytes/s, I/O bus bandwidth is 100 MBytes/s, network interface occupancy per packet is 1000 processor cycles and total interrupt cost is 1000 processor cycles.

Parameter	Range	Achievable	Best
Host Overhead (cycles)	0-10000	600	~0
I/O Bus Bandwidth (MBytes/MHz)	0.25-2	0.5	2
NI Occupancy (cycles)	0-10000	1000	200
Interrupt Cost(cycles)	0-50000	500	~0

Table 4.1: Ranges and achievable and best values of the communication parameters under consideration.

4.2 Effects of Communication Parameters

Application	Host Ovrhd	NI Occup	I/O b/w	Intrpt Cost	Page Size	Procs/Node
FFT	22.6%	11.9%	40.8%	86.6%	72.6%	13.8%
LU-cont	17.9%	7.5%	15.9%	70.8%	34.4%	-35.3%
Ocean-cont	4.5%	2.8%	6.5%	35.2%	19.6%	63.2%
Water-nsq	32.4%	16.6%	10.8%	83.2%	62.2%	-87.1%
Water-spa	23.7%	8.5%	8.9%	67.9%	51.0%	-87.5%
Radix	35.8%	-31.8%	77.6%	58.7%	-368.2%	-699.4%
Volrend	34.7%	12.8%	15.7%	91.3%	63.9%	-68.1%
Raytrace	8.2%	2.9%	8.9%	52.3%	9.1%	-16.1%
Barnes-reb	40.7%	21.8%	44.8%	80.3%	71.5%	-383.4%
Barnes-spa	4.4%	-0.6%	27.5%	59.0%	-109.6%	-49.4%

Table 4.2: Maximum Slowdowns with respect to the various communication parameters for the range of values with which we experiment. Negative numbers indicate speedups.

In this section we present the effects of each parameter on the performance of an all–software HLRC-SMP protocol for a range of values. Table 4.2 presents the maximum slowdowns for each application for the parameters under consideration. The maximum slowdown is computed from the speedups for the smallest and biggest values considered for each parameter, keeping all other parameters at their achievable values.

Negative numbers indicate speedups. The rest of this section discusses the parameters one by one. For each parameter we also identify the application characteristics that most closely predict the effect of that parameter. The next section will take a different cut, looking at the bottlenecks on a per–application rather than per–parameter basis. At the end of this section we also present results for AURC-SMP.

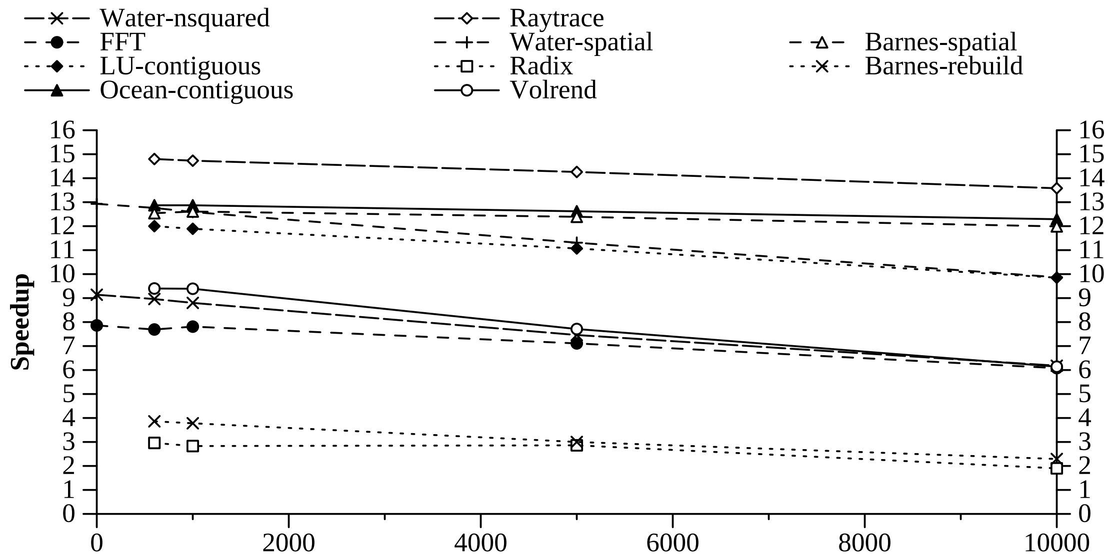

Figure 4.1: Effects of host overhead on application performance. The data points for each application correspond to a host overhead of 0, 600, 1000, 5000, and 10000 processor cycles.

Host overhead: Figure 4.1 shows that the slowdown due to the host overhead is generally low, especially for realistic values of asynchronous message overheads. However, it varies among applications from less than 10% for Barnes-spatial, Ocean-contiguous and Raytrace to more than 35% for Volrend, Radix and Barnes-rebuild across the entire range of values. In general, applications that send more messages exhibit a higher dependency on the host overhead. This can be seen in Figure 4.2, which shows two curves. One is the slowdown of each application between the smallest and highest host overheads that we simulate, normalized to the biggest of these

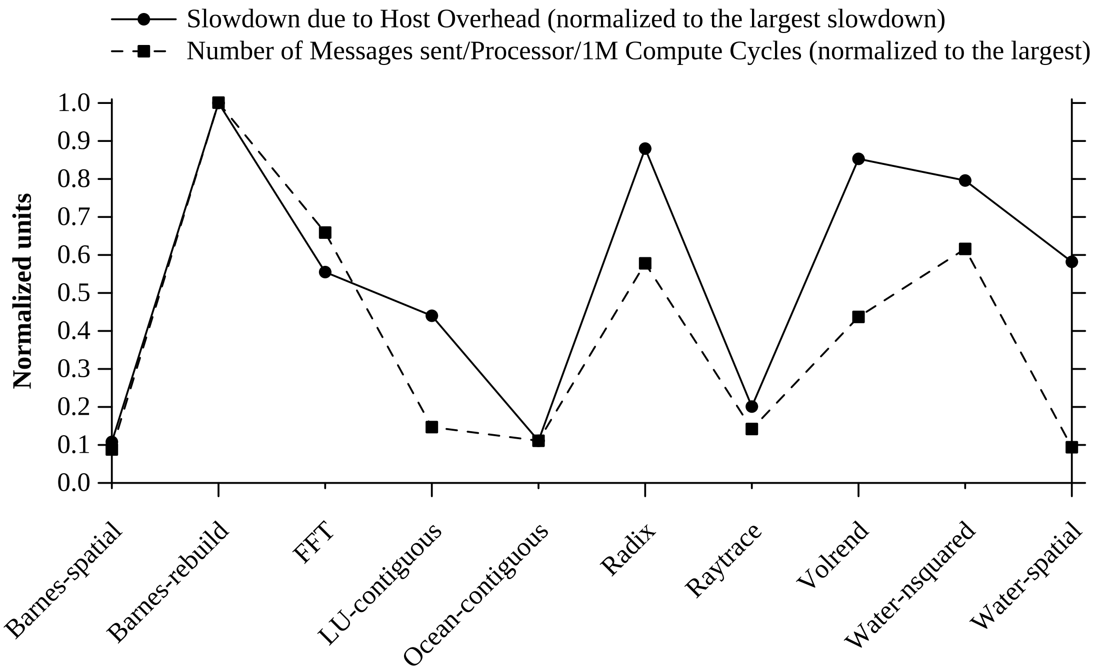

Figure 4.2: Relation between slowdown due to Host Overhead and Number of Messages sent.

slowdowns. The second curve is the number of messages sent by each processor per–10^6 compute cycles, normalized to the biggest of these numbers of messages. Note that with asynchronous messages, host overheads will be on the low side of our range, so we can conclude that host overhead for sending messages is not a major performance factor for coarse grain SVM systems and is unlikely to become so in the near future.

Network interface occupancy: Figure 4.3 shows that network interface occupancy has even a smaller effect than host overhead on performance, for realistic occupancies. Most applications are insensitive to it, with the exception of a couple of applications that send a large number of messages. For these applications, slowdowns of up to 22% are observed at the highest occupancy values. The speedup observed for Radix is in reality caused by timing issues (contention is the bottleneck in Radix).

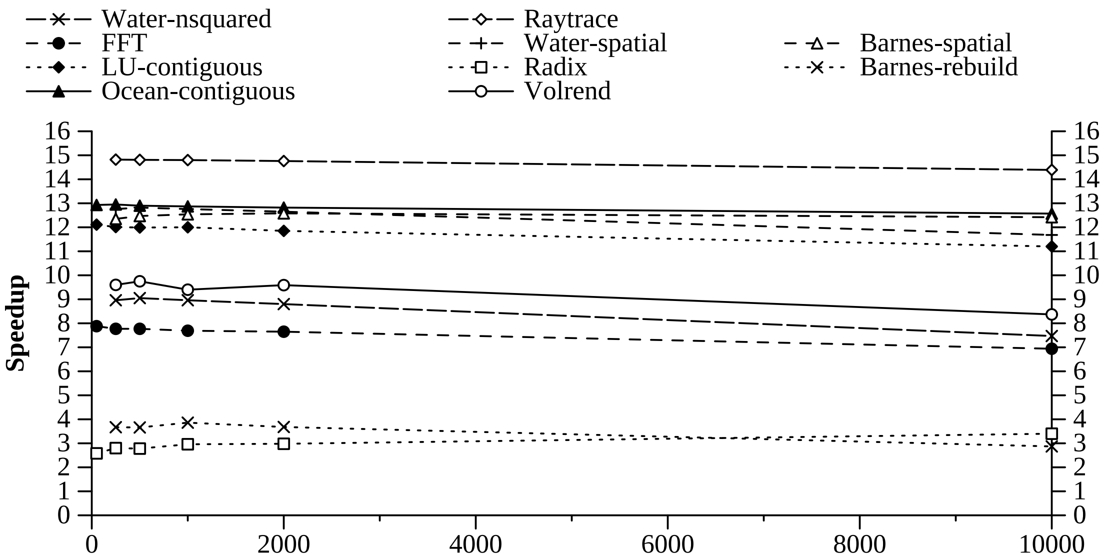

Figure 4.3: Effects of network interface occupancy on application performance. The data points for each application correspond to a network occupancy of 50, 250, 500, 1000, 2000, and 10000 processor cycles.

I/O bus bandwidth: Figure 4.4 shows the effect of I/O bandwidth on application performance. Reducing the bandwidth results in slowdowns of up to 82%, with 4 out of 11 applications exhibiting slowdowns of more than 40%. However, many other applications are not so dependent on bandwidth, and only FFT, Radix, and Barnes-rebuild benefit much from increasing the I/O bus bandwidth beyond the achievable relationships to processor speed today. Of course, this does not mean that it is not important to worry about improving bandwidth. As processor speed increases, if bandwidth trends do not keep up, we will quickly find ourselves at the relationship reflected by the lower bandwidth case we examine (or even worse). What it does mean is that if bandwidth keeps up with processor speed, it is not likely to be the major limitation on SVM systems for applications.

Figure 4.5 shows the dependency between bandwidth and the number of bytes sent per processor for each application. As before, units are normalized to the maximum of the numbers presented for each curve. Applications that exchange a lot of data,

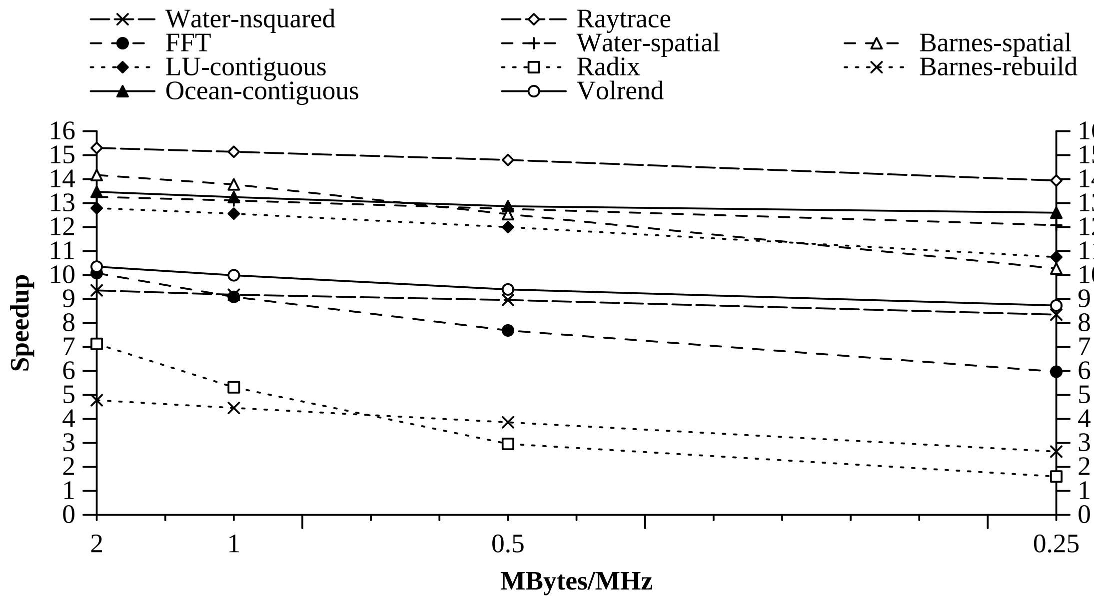

Figure 4.4: Effects of I/O bandwidth on application performance. The data points for each application correspond to an I/O bandwidth of 2, 1, 0.5 and 0.25 MBytes per–processor clock MHz, or 400, 200, 100, and 50 MBytes/s assuming a 200 MHz processor.

not necessarily a lot of messages, need higher bandwidth.

Interrupt cost: Figure 4.6 shows that interrupt cost is a very important parameter in the system. Unlike bandwidth, it affects the performance of *all* applications dramatically, and in many cases a relatively small increase in interrupt cost leads to a big performance degradation. For most applications, interrupt costs of up to about 2000 processor cycles for each of initiation and delivery do not seem to hurt much. However, commercial systems typically have much higher interrupt costs. Increasing the interrupt cost beyond this point begins to hurt sharply. All applications have a slowdown of more than 50% when the interrupt cost varies from 0 to 50000 processor cycles (except Ocean-contiguous that exhibits an anomaly since the way pages are distributed among processors changes with interrupt cost). This suggests that architectures and operating systems should work harder to improving interrupt costs

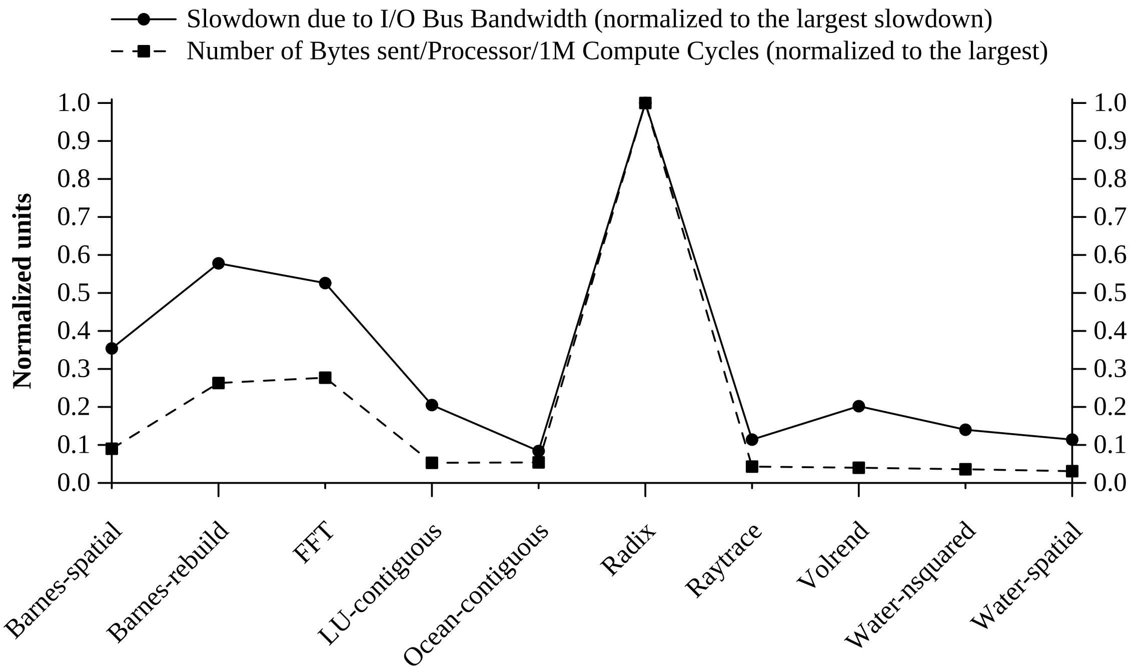

Figure 4.5: Relation between slowdown due to I/O Bus Bandwidth and Number of Bytes transferred.

if they are to support SVM well, and SVM protocols should try to avoid interrupts as much as possible, Figure 4.7 shows that the slowdown due to the interrupt cost is closely related to the number of protocol events that cause interrupts—page fetches and remote lock acquires.

With SMP nodes there are many options for how interrupts may be handled within a node. Our protocol uses one particular method. Systems with uniprocessor nodes have less options, so we experimented with such configurations as well. We found that interrupt cost is important in that case as well. The only difference is that the system seems to be a little less sensitive to interrupt costs of between 2500 and 5000 cycles. After this range, performance degrades quickly as in the SMP configuration.

We also experimented with round robin interrupt delivery and the results look similar to the case where all interrupts are delivered to a fixed processor in each SMP. Overall performance seems to increase slightly, compared to the static interrupt

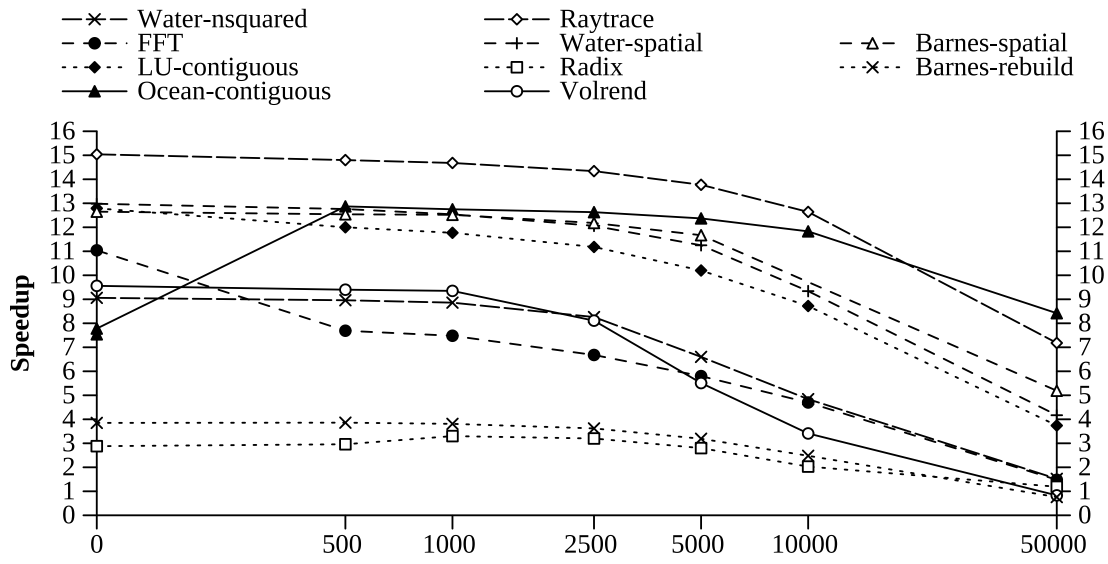

Figure 4.6: Effects of interrupt cost on application performance. The six bars for each application correspond to an interrupt cost of 0, 500, 1000, 2500, 5000, 10000, and 50000 processor cycles.

scheme, but as in the static scheme it degrades quickly as interrupt cost increases. Moreover implementing such a scheme in a real system may be complicated and may incur additional costs.

AURC-SMP: As mentioned in the introduction, besides HLRC-SMP, we also used AURC-SMP to study the effect of the communication parameters when using hardware support for automatic write propagation instead of software diffs. The results look very similar to HLRC-SMP, with the exception that network interface occupancy is much more important in AURC-SMP. The automatic update mechanism may generate more traffic through the network interface because new values for the same data may be sent multiple times to the home node before a release. More importantly, the number of packets may increase significantly since updates are sent at a much finer granularity, so if they are apart in space or time they may not be coalesced well into packets. Figure 4.8 shows how performance changes as the NI overhead increases for

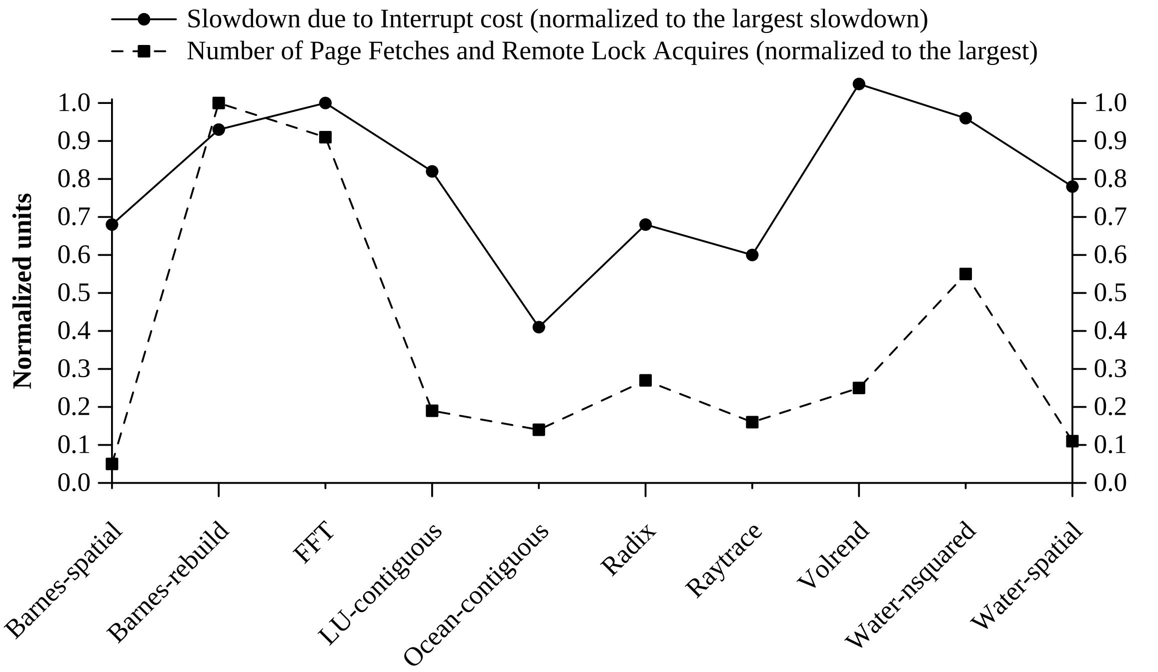

Figure 4.7: Relation between slowdown due to Interrupt cost and Number of Page Fetches and Remote Lock Acquires.

both regular and irregular applications.

4.3 Limitations on Application Performance

In this section we examine the difference in performance between the *best* configuration and an ideal system (where the speedup is computed only from the compute and local stall times, ignoring communication and synchronization costs), and the difference in performance between the *achievable* and the *best* configuration on a per–application basis. Recall that best stands for the configuration where all communication parameters assume their best value, and achievable stands for the configuration where the communication parameters assume their achievable values. The goal is to identify the application properties and architectural parameters that are responsible for the difference between the best and the ideal performance, and the parameters

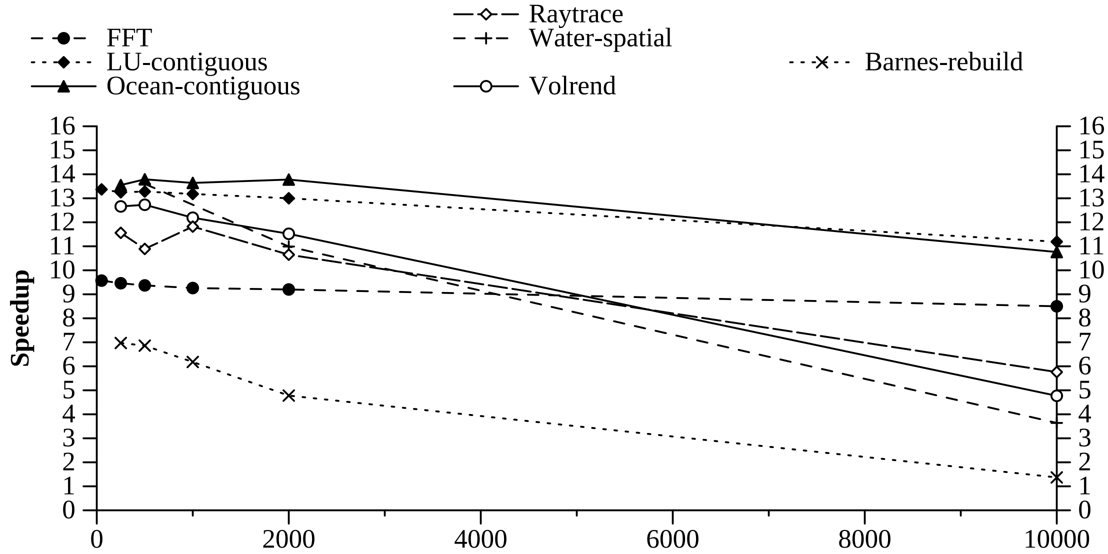

Figure 4.8: Effects of network interface occupancy on application performance for AURC-SMP. The data points for each application correspond to a network occupancy of 50, 250, 500, 1000, 2000, and 10000 processor cycles.

that are responsible for the difference between the achievable and the best performance. The speedups for each configuration will be called *ideal, best* and *achievable* respectively. Table 4.3 shows these speedups for all applications. In many cases, the achievable speedup is close to the best speedup. However, in some cases (FFT, Radix, Barnes-rebuild) there remains a gap. The performance with the best configuration is often quite far from the ideal speedup. To understand these effects, let us examine each application separately.

FFT: The best speedup for FFT is about 13.5. The difference from the ideal speedup of 16.2 comes from data wait time at page faults, which have a cost even for the best configuration, despite the very high bandwidth and the zero–cost interrupts. The achievable speedup is about 7.7. There are two major parameters responsible for this drop in performance: the cost of interrupts and the bandwidth of the I/O bus. Making the interrupt cost 0 results in a speedup of 11, while increasing the I/O

Application	Best	Achievable	Ideal
FFT	13.5	7.7	16.2
LU-contiguous	13.7	14.0	18.9
Ocean-contiguous	10.5	13.0	16.0
Water-nsquared	9.9	9.0	15.8
Water-spatial	13.7	13.3	15.8
Radix	7.0	3.0	16.1
Volrend	10.9	9.40	15.4
Raytrace	15.6	14.8	16.4
Barnes-rebuild	5.9	3.9	15.4
Barnes-spatial	14.5	12.5	15.6

Table 4.3: Best and Achievable Speedups for each application

bus bandwidth to the memory bus bandwidth gives a speedup of 10. Modifying both parameters at the same time gives a speedup almost the same as the best speedup.

LU-contiguous: The best speedup is 13.7. The difference from the ideal speedup is due to load imbalances in communication and due to barrier cost. The achievable speedup for LU-contiguous is about the same as the best speedup, since this application has very low communication to computation ratio, so communication is not the problem.

Ocean-contiguous: The best speedup for Ocean-contiguous is 10.55. The reason for this is that when the interrupt cost is 0 an anomaly is observed in first touch page allocation and the speedup is very low due to a large number of page faults. The achievable speedup is 13.0, with the main cost being that of barrier synchronization. It is worth noting that speedups in Ocean-contiguous are artificially high because of local cache effects: a processor's working set does not fit in cache on a uniprocessor, but does fit in the cache with 16 processors. Thus the sequential version performs poorly due to the high cache stall time.

Barnes-rebuild: The best speedup for Barnes-rebuild is 5.90. The difference from the ideal is because of page faults in the large number of critical sections (locks). The achievable speedup is 3.9. The difference between the best and achievable speedups in the presence of page faults is because synchronization wait time is even higher due to the increased protocol costs. These increased costs are mostly because of the host overhead (a loss of about 1 in the speedup) and the NI occupancy (about 0.8). To verify all these we disabled remote page fetches in the simulator so that all page faults appear to be local. The speedup becomes 14.64 in the best and 10.62 in the achievable cases respectively. The gap between the best and the achievable speedups is again due to host and NI overheads.

Barnes-spatial: The second version of Barnes we run is an improved version with minimal locking [47]. The best speedup is 14.5, close to the ideal. The achievable speedup is 12.5. The difference between these two is mainly because of the lower available I/O bandwidth in the achievable case. This increases the data wait time in an imbalanced way.

Water-nsquared: The best speedup for Water-nsquared is 9.9 and the achievable speedup is about 9. The reason for the not very high best speedup is page faults that occur in contended critical sections, greatly increasing serialization at locks. If we artificially disable remote page faults the best speedup increases from 9.9 to 14.1. The cost for locks in this artificial case is very small and the non-ideal speedup is due to imbalances in the computation itself.

Water-spatial: The best speedup is 13.75. The difference from ideal is mainly due to small imbalances in the computation and lock wait time. Data wait time is very small. The achievable speedup is about 13.3.

Radix: The best speedup for Radix is 7. The difference from the ideal speedup of 16.1 is due to data wait time, which is exaggerated by contention even at the *best* parameter values, and the resulting imbalances among processors which lead to high synchronization time. The imbalances are observed to be due to contention in the network interface. The achievable speedup is only 3. The difference from the best speedup is due to the same factors: data wait time is much higher and much more imbalanced due to much greater contention effects. The main parameter responsible for this is I/O bus bandwidth. For instance, if we quadruple I/O bus bandwidth the achievable speedup for Radix becomes 7, just like the best speedup.

Raytrace: Raytrace performs very well. The best speedup is 15.64 and the achievable speedup 14.80.

Volrend: The best speedup is 10.95. The reason for this low number is imbalances in the computation itself due to the cost of task stealing, and large lock wait times due to page faults in critical sections. If we artificially eliminate all remote page faults, then computation is perfectly balanced and synchronization costs are negligible (speedup is 14.9 in this fictional case). The achievable speedup is 9.40, close to the best speedup.

We see that the difference between ideal and best performance is due to page faults that occur in critical sections, I/O bandwidth limitations and imbalances in the communication and computation, and the difference between best and achievable performance is primarily due to the interrupt cost and I/O bandwidth limitations and less due to the host overhead. Overall, application performance on SVM systems today appears to be limited primarily by interrupt cost, and next by I/O bus bandwidth. Host overhead and NI occupancy per packet are substantially less significant, and in that order.

4.4 Page Size and Degree of Clustering

In addition to the performance parameters of the communication architecture discussed above, the granularities of coherence and data transfer–i.e. the page size–and the number of processors per node are two other important parameters that affect the behavior of the system. They play an important role in determining the amount of communication that takes place in the system, the cost of which is then determined by the performance parameters.

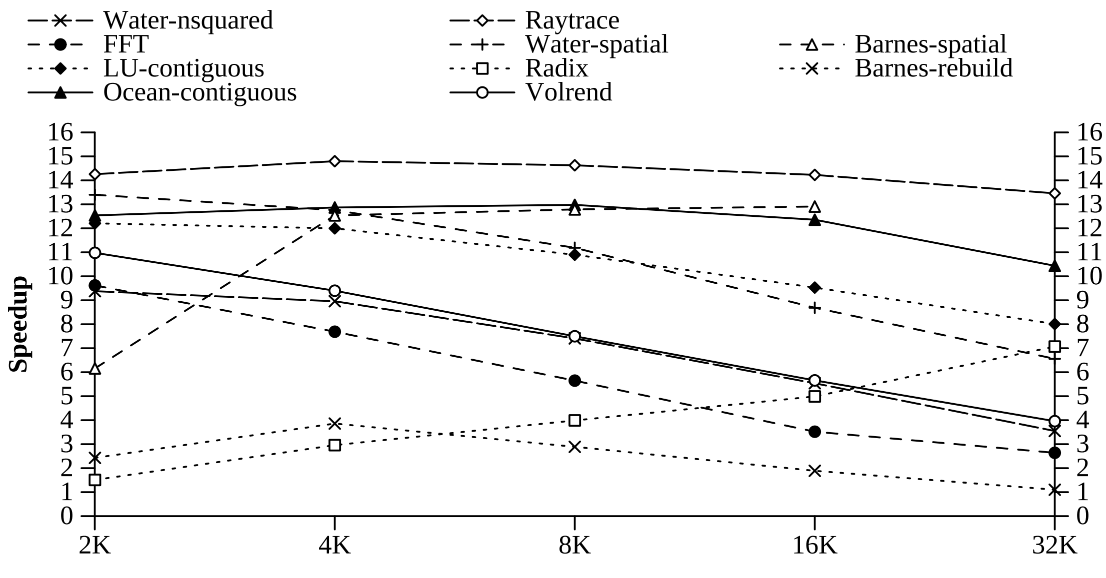

Figure 4.9: Effects of page size on application performance. The data points for each application correspond to a page size of 2 KBytes, 4 KBytes, 8 KBytes, 16 KBytes, and 32 KBytes.

Page size: The page size in the system is important for many reasons. It defines the size of the transfers, since in all software protocols data fetches are performed at page sizes. It also affects the amount of false sharing in the system, which is very important for SVM. These two aspects of the page size conflict with each other: bigger pages reduce the number of messages in the system if spatial locality is well exploited in communication, but they increase the amount of false sharing, and vice versa. Moreover, different page sizes lead to different amounts of fragmentation in memory, which may result in wasted resources. Figure 4.9 shows that the effects of page size on applications vary a lot. Most applications seem to favor smaller page sizes, with the exception of Radix that benefits a lot from bigger pages. We vary the page size between 2 KBytes and 32 KBytes pages. Most systems today support either 4 KBytes or 8 KBytes pages. We should note two caveats in our study with respect to page size. First, we did not tune the applications specifically to the different page sizes. Second, the effects of the page size are often related to the problem sizes that are used. For applications in which the amount of false sharing and fragmentation (i.e. the granularity of access interleaving in memory from different processors) changes with problem size, larger problems that run on real systems may benefit from larger pages (i.e. FFT).

Cluster size: The degree of clustering is the number of processors per node. Figure 4.10 shows that for most applications greater clustering helps even if the memory configuration and bandwidths are kept the same [1]. We use cluster sizes of 1, 4, 8 and 16 processors, always keeping the total number of processors in the system at 16. These configurations cover the range from a uniprocessor node configuration to a cache–coherent, bus–based multiprocessor. Typical SVM systems today use either uniprocessor or 4–way SMP nodes. A couple of interesting points emerge. First, unlike most applications, for Ocean-contiguous the optimal clustering is four proces-

[1] This assumption, of keeping the memory subsystem the same and increasing the number of processors per node is not very realistic, since systems with higher degrees of clustering usually have a more aggressive memory subsystem as well, and are likely to provide greater node–to–network bandwidth.

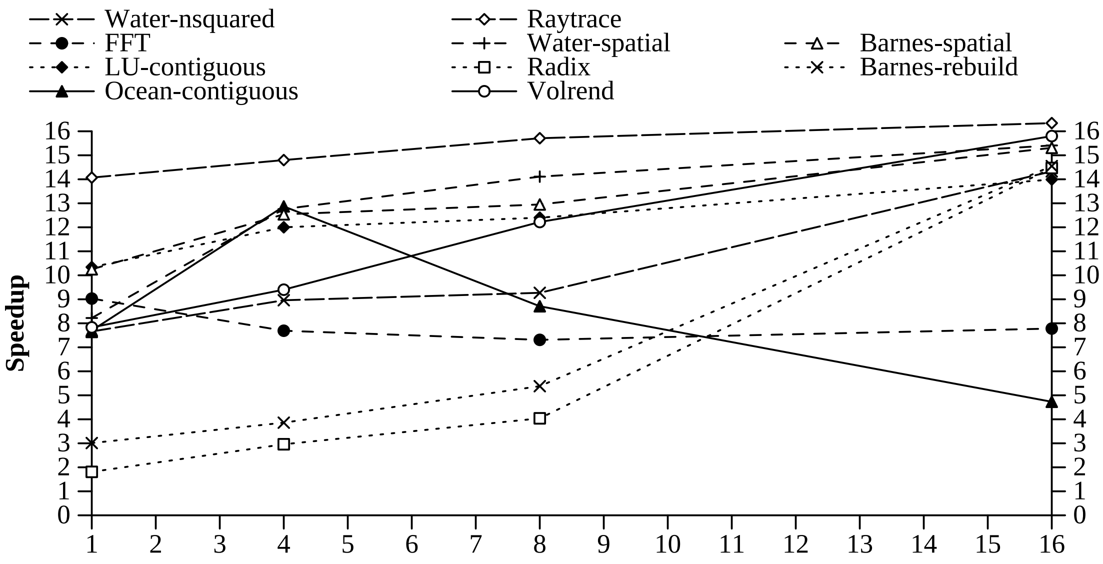

Figure 4.10: Effects of cluster size on application performance. The data points for each application correspond to a cluster size of 1, 4, 8, and 16 processors per node.

sors per node. The reason is that Ocean-contiguous generates a lot of local traffic on the memory bus due to capacity and conflict misses, and more processors on the bus exacerbate this problem. On the other hand, Ocean-contiguous benefits a lot from clustering because of the communication pattern. Thus when four processors per node are used, the performance improvement over one processor per node comes from sharing. When the system has more than four processors per node, the memory bus is saturated and although the system benefits from sharing, performance is degrading because of memory bus contention. Radix and FFT also put greatly increased pressure on the shared bus. The cross–node SVM communication however, is very high and the reduction in it via increased spatial locality at page grain due to clustering outweighs this problem. The second important point is that the applications that perform very poorly under SVM do very well on a shared bus system at this scale. The reason is that these applications either exhibit a lot of synchronization or make fine grain accesses, both of which are much cheaper on a hardware–coherent shared bus architecture. For example, applications where the problem in SVM is page faults within critical sections (i.e. Barnes-rebuild) perform much better on this architecture. These results show that bus bandwidth is not the most significant problem for these applications at this scale, and the use of hardware coherence and synchronization outweighs the problems of sharing a bus.

In Chapter 3 we discussed the protocol and performance implications for extending HLRC to support small–scale SMPs (HLRC-SMP). Also, Section 3.5 presents statistics about the applications for systems with 1,4 and 8 processors per node. To further explore the effects of clustering with our protocol we examine the performance of the system when the number of processors per node varies from 1 to 4 to 8, always keeping the total number of processors at 16. An important problem that occurs with clustering is that the useful computation within a node speeds up faster than the communication across nodes is reduced, putting more stress on the now shared note–to–network bandwidth. We conduct two experiments. In the first we keep I/O bandwidth constant at 100 MBytes/s as the number of processors increase, whereas in the second we also increase the I/O and node–to–network bandwidth with cluster size from 100 MBytes/s to 200 MBytes/s to 400 MBytes/s. Figure 4.11 shows how the speedup varies with cluster size. Each application has three bars that correspond to a configuration with 1,4 and 8 processors per node. The white bars for each application represent speedups as the cluster size increases and the bandwidth remains constant to 100 MBytes/s. The black additions to each bar represent the additional speedup gain as bandwidth increases with cluster size from 100 MBytes/s to 200 MBytes/s to 400 MBytes/s. In cases where applications had poor performance on uniprocessor systems, using SMPs helps substantially. Especially, if bandwidth is increased as well, the performance of applications that are traditionally difficult for SVM (i.e. Radix) becomes decent [2].

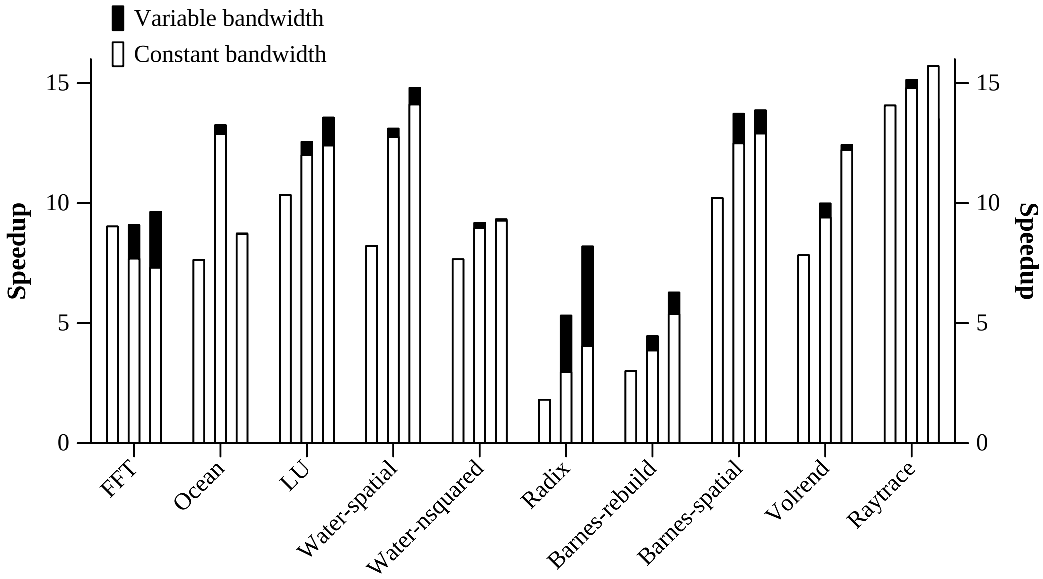

Figure 4.11: Applications speedups for HLRC-SMP with increasing cluster size.

In conclusion, we find that, when comparing 4–way SMP nodes versus uniprocessor nodes for the same total processor count (through detailed architectural simulations), performance improves substantially with the use of SMPs in six out of ten very different applications. The other four show a smaller improvement. All applications improve further when 8–way SMP nodes are used, except Ocean-contiguous where the problem becomes saturation of the memory bus inside each node. Among the irregular applications we examine, Barnes-rebuild, Radix, Volrend and Water-spatial benefit substantially and Barnes-spatial, Raytrace and Water-nsquared do not exhibit any significant improvement.

[2]For Ocean-contiguous the best case is always four processors per node. The reason for this is the communication pattern of nearest neighbor and the limited memory bus bandwidth. If the number of processors per node is less than four, then protocol costs are high, because of the communication pattern. If the cluster size is bigger than four, then, although protocol costs do not increase, the contention on the memory bus becomes a problem reducing performance.

4.5 Related Work

Our work is similar in spirit to some earlier studies, conducted in [64, 41], but in different context. In [64], the authors examine the impact of communication parameters on end performance of a network of workstations with the applications being written in Split-C on top of Generic Active Messages. They find that application performance demonstrates a linear dependence on host overhead and on the gap between transmissions of fine grain messages. For SVM, we find these parameters to not be so important since their cost is usually amortized over page granularity. Applications were found to be quite tolerant to latency and bulk transfer bandwidth in the split-C study as well.

In [41], Holt et al. find that the occupancy of the communication controller is critical to good performance in DSM machines that provide communication and coherence at cache line granularity. Overhead is not so significant there (unlike in [64]) since it is very small.

In [49], Karlsson et al. find that the latency and bandwidth of an ATM switch is acceptable in a clustered SVM architecture. In [53] a Lazy Release Consistency protocol for hardware cache–coherence is presented. In a very different context, they find that applications are more sensitive to the bandwidth than the latency component of communication.

Several studies have also examined the performance of different SVM systems across multiprocessor nodes and compared it with the performance of configurations with uniprocessor nodes. Erlichson et al. [28] find that clustering helps shared memory applications. Yeung et al. in [99] find this to be true for SVM systems in which each node is a hardware coherent DSM machine. In [9], they find that the same is true in general for all software SVM systems, and for SVM systems with support for automatic write propagation.

4.6 Discussion and Conclusions

This work shows that there is room for improving SVM cluster performance in various directions:

- Interrupts. Since reducing the cost of interrupts in the system can improve performance significantly, an important direction for future work is to design SVM systems that reduce the frequency and/or the cost of interrupts. Polling, better operating system support, or support for remote fetches that do not involve the remote processor are mechanisms that can help in this direction. Operating system and architectural support for inexpensive interrupts would improve system performance. Unfortunately this is not always achieved, especially in commercial systems. In these cases, protocol modifications (using non–interrupting remote fetch operations) or implementation optimizations (using polling instead of interrupts) can improve system performance and lead to more predictable and portable performance across different architectures and operating systems. Polling can be done either by instrumenting the applications or (in SMP systems) by reserving one processor for protocol processing. Recent results for interrupts versus polling in SVM systems vary. One study finds that polling may add a significant overhead, leading to inferior performance than interrupts for page grain SVM systems [101]. On the other hand, Stets et al. find that polling gives generally better results than interrupts [91]. We believe more research is needed on modern systems to understand the role of polling. Another interesting direction that we are exploring is moving some of the protocol processing itself to the network processor found in programmable network interfaces like such as Myrinet, thus reducing the need for interrupting the main processor.

- System bandwidth. Providing high bandwidth is also important, to keep up with increasing processor speeds. Although fast system interconnects are available, software performance is, in practice, rarely close to what the hardware provides. Low level communication libraries fail to deliver close to raw hardware performance in many cases. Further work on low level communication interfaces may also be helpful in providing low–cost, high–performance SVM systems. Multiple network interfaces per node is another approach that can increase the available bandwidth. In this case protocol changes may be necessary to ensure proper event ordering.

- Clustering. Up to the scale we examined, adding more processors per node helps in almost all cases. In applications where performance does not increase quickly with the cluster size, scaling of other system parameters, such as memory bus and I/O bandwidth, can have the desirable effects.

- Applications. In doing this work we found that restructuring applications is an area that can make a big difference. Understanding how an application behaves and restructuring it properly can dramatically improve performance far beyond the improvement in system parameters or protocols [47]. This however, is not always easy, and, unfortunately, not many tools are available in parallel systems to help easily discover the cause of bottlenecks and obtain insight about application restructuring needs, especially when contention is a major problem as it often is in commodity–based communication architectures. Architectural simulators are one of the few tools that can currently be used to understand how an application behaves in detail.

We should point out that this work is limited to a certain family of home–based SVM protocols. Other systems—for instance fine grain SVM systems—may exhibit different behavior and dependencies on communication parameters. Similar studies for other protocols and architectures can help us understand better the differences and similarities among SVM systems.

Application	PFaults	PFetches	Rem Acq	Loc Acq	Bar	MB Sent	Msgs Sent
FFT	1.46	1.70	-	-	2.00	1.75	1.70
LU-cont	1.94	2.53	1.86	2.00	1.90	12.90	3.66
Ocean-cont	0.75	0.53	2.77	1.57	1.99	2.50	1.95
Water-nsq	2.89	2.63	1.40	2.50	1.99	2.80	2.37
Water-spa	1.85	2.05	1.68	2.26	1.98	2.00	2.08
Radix	1.83	2.43	2.70	4.10	1.99	2.19	2.38
Volrend	1.45	1.79	-	-	1.98	1.90	1.35
Raytrace	2.08	2.08	1.33	2.40	2.00	2.08	1.83

Table 4.4: Ratios of protocol events for a 32 and a 16 processor configuration (4 processor per node). The high ratio for LU is due to the page allocation policy that depends on the number of processors. With 32 processors in the system, pages are not allocated properly and there is a lot of traffic in the system due to diffs.

This work was based on a 16 processor system. To address the question of what happens in bigger systems we run some experiments with a 32 processor configuration and compared the number of protocol events between the two configurations. Table 4.4 shows the ratios of protocol events and communication traffic between a 32 and a 16 processor configuration. In most cases the event counts scale proportionally with the size of the system which leads us to believe that the results presented so far will hold for bigger configurations as well (at least up to 32 processors). Moreover, with larger problem sizes the problems related to the communication architecture are usually alleviated. However, more sophisticated scaling models, that take into account the problem size, may be necessary for more detailed and accurate predictions.

Another important question is how are these communication parameters going to scale with time. It seems that the parameters that closely follow hardware performance (host overhead, network interface occupancy, bandwidth) have more potential for getting better (relative to processor speeds) than interrupt cost which depends on the operating system and on special architectural support.

In conclusion, we have examined the effects of communication parameters to a family of SVM protocols. Through detailed architectural simulations of a cluster of SMPs and a variety of applications, we find that most applications are very sensitive to interrupt cost, and a few would benefit from improvements in bandwidth relative to processor speed as well. Unbalanced systems with relatively high interrupt costs and low I/O bandwidth can result in substantial losses in application performance. In these cases we observe slowdowns of more than 90% (a factor of 10 longer execution time). However, most applications are not sensitive to host overhead and network interface occupancy.

Most regular applications can achieve very good SVM performance under the *best* configuration of parameters. For irregular applications, though, even this best performance can be low. This is mainly due to serialization effects in critical sections, i.e. due to page faults incurred inside critical sections, which dilate the critical sections and increase serialization. For example by reducing the amount of locking by using a different algorithm for parallel tree building, the performance of Barnes improves by a factor of 2-3.

Overall, the achievable application performance today is limited primarily by interrupt cost and then by node to network bandwidth. Host overhead and NI occupancy appear less important to improve relative to processor speed. If interrupts are free and bandwidth high relative to the processor speed, then the achievable performance approaches the best performance in most cases.

Chapter 5

Network Interface Support for SVM

Chapter 4 explored the relationship of SVM application performance to the underlying communication architecture and revealed the significant system bottlenecks. This chapter investigates the use of mechanisms in the software communication layer and the underlying network interface to alleviate these bottlenecks and substantially enhance the performance of shared virtual memory (SVM) on clusters of SMPs. To demonstrate the benefits of this approach we use a real implementation with a programmable network interface as our prototype, but our extensions are general–purpose and can be provided by network interfaces that do not employ a programmable processor.

Previous work has studied several SVM systems and protocols through both implementation and simulation [29, 44, 54, 7, 45, 100, 9, 10, 75], and has shown that the use of SMP nodes indeed improves performance for software shared memory protocols [49, 9, 91, 78]. In all these cases however, speedups on a 16–processor system were still found to be substantially lower than those expected from the programs on hardware–coherent machines. Figure 5.1 shows the comparison between an efficient hardware–coherent machine and the all–software HLRC-SMP [75] system for several applications; the processors used by the two systems are different, so a direct comparison cannot be made, but the qualitative result is clear. Thus, it is important to develop techniques for improving SVM performance on clusters.

We examine how the protocol layer can take advantage of each mechanism in the communication layer and be restructured accordingly. The final protocol (SVM-NI) eliminates the need for interrupts and asynchronous protocol handling. For each mechanism, we evaluate the impact on the end performance of ten applications with widely varying characteristics. We demonstrate that substantial improvements in performance can indeed be achieved, and find that different applications need different mechanisms among the ones we use. Application performance improves by (min -20.16%, avg 37.68%, max 117.6%) across all applications. individual components of execution time targeted by each mechanism are reduced by even higher percentages. Finally, we use a firmware performance monitor [60], integrated with the communication layer, to understand the drawbacks of the system, to identify interesting tradeoffs in the protocol layer for future exploration, and to identify the remaining bottlenecks in SVM performance that should be addressed next.

5.1 Approach and Related Work

As shown in Figure 5.1, for many applications the performance of shared virtual memory (SVM) on clusters is far from the performance that hardware DSM systems provide. This is despite the fact that system area networks (SANs) [17, 37, 43] provide increasingly low latency (on the order of a few microseconds today) and high bandwidth (on the order of 100 MBytes/s) in hardware, and communication

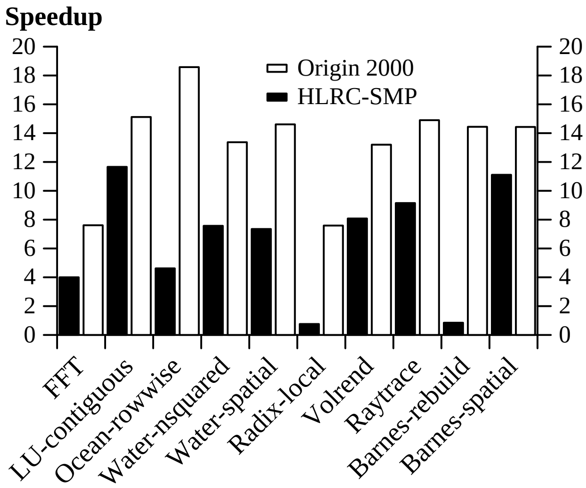

Figure 5.1: Application speedups for a hardware DSM machine (Origin–2000) and an SVM system (our base protocol).

software delivers almost all of this performance to user programs. However, network interfaces also implement increased functionality compared to traditional local area networks (LAN), which can be exploited to enhance SVM performance. For example, the network interface in SHRIMP [16] and the DEC Memory Channel [37] provide some form of automatic propagation of writes to remote memory, which is studied in Chapter 3. Furthermore, SAN networks like Myrinet have started to provide programmable processors on the network interface that can be used to implement special functionality [17].

The capabilities of SAN network interfaces can be taken advantage of in several ways:

1. To improve the performance of traditional send and receive operations.

2. To provide support for general–purpose data movement and synchronization mechanisms that can be utilized by protocols to alleviate key bottlenecks.

3. To perform protocol processing itself in the network interface.

The first type of support has been exploited in many SVM projects and is also used in our base system, HLRC-SMP via the VMMC communication layer. At the other end of the spectrum, the network interface can be used to perform full–blown protocol processing, including diff creation and application and the management of time–stamps and write notices. This approach requires that (i) the NI be fast enough relative to the main processor at performing protocol operations, (ii) the NI be able to access the application data and the protocol data quickly and safely, (iii) the main processor and the NI have a consistent view of data mappings and protections. Typically, this approach requires a fast communication coprocessor with similar access to main memory as the main processor (e.g., on the memory bus instead of in the network interface). The Intel Paragon provided such a communication architecture, and this approach was taken for SVM in [100]; however, the performance benefit in that case was found to be small. SANs that include a programmable processor in the NI itself usually employ much slower processors (in terms of both clock rate and the architectural features) than the host processor and do not have good fine–grained access to the host's main memory. Moreover, in cases where a fast, symmetric coprocessor is available on the node, the cost of protocol processing does not justify devoting a whole fast processor to it, and that processor is likely to be better used as an additional processor for application computation within the node.

Our focus here is on the intermediate approach: taking advantage of general purpose mechanisms for data movement and synchronization in the network interface to reduce key costs and reduce the involvement of the main processors. Such mechanisms include support for:

- automatic (implicit) transfer of application and protocol data based on memory loads and stores (as in AURC for stores)

- reducing the cost of explicit communication used in protocols and processor involvement in protocol handling
- providing protocol–independent synchronization functionality in the network interface.

Previous work [44, 54, 7] has partially addressed the first of these possibilities by taking advantage of write propagation support in the network interface to propagate shared or protocol data to remote memories without involving the remote processors. AURC [44] that takes advantage of write propagation support was discussed in Chapter 3. Similarly, the Cashmere protocol [54] takes advantage of the remote write capability of the DEC Memory Channel interconnect to propagate updates somewhat differently in the context of a different, more eager protocol. [7] proposes the use of a hardware diff engine to propagate updates to shared data. A discussion on how remote write access capabilities of VM–based networks can be used in SVM systems is provided in [39]. In all the above cases, any other necessary protocol and data movement activity, e.g. page fetches, etc. is handled through either interrupts or code instrumentation.

As discussed in Chapter 3, the automatic write propagation used by AURC requires either that second–level caches be write–through or that substantial intrusive hardware support be provided to ensure that the appropriate modifications from a write–back cache are indeed propagated according to the consistency model [10]. Modern nodes do not provide either type of support. The Cashmere system uses the DEC Memory Channel network and interface, which has a memory–mapped I/O address space. Writes that need to be propagated have to be instrumented so that they are *doubled* to this address space as well, whereupon they are automatically propagated by the NI. Myrinet and many other general interfaces do not provide this automatic propagation support. Moreover, we saw in Chapter 3 that the gain from providing this support and hence avoiding diffs was not large [10].

We do not use this approach of data propagation based on ordinary or instrumented stores (or even loads) in this work. Rather, we focus on the second and third possibilities listed above: Internode communication is triggered by explicit data or protocol messages (e.g., to update a page with a diff, to explicitly fetch a page, etc.) issued by a processor or by explicit synchronization operations, not directly by load and store operations to memory (e.g. to propagate the updates as a result of a store instruction, to fetch remote data implicitly as a result of a local read instruction, etc.), and the NI is used to support these operations better and to reduce the involvement of compute processors. Neither of these cases has been explored before to our knowledge.

5.2 HLRC-SMP implementation

Chapter 3 discussed the issues that arise in extending the HLRC protocol to support SMP nodes efficiently. In this section we present the actual implementation of HLRC-SMP on a cluster of SMPs. We discuss a few key implementation issues that we encountered and found to be important. This section presents the choices that were made in the implementation of these protocols in the actual system and the data structures that allow us to easily implement the above choices in the HLRC-SMP, as well as how the synchronization and page fault operations are managed using them.

Data structures: The data structures used in the implementation are very similar, but not identical, to the data structures used in the simulator. In the implementation of the protocol on the actual system the bins data structure keeps the modified pages for each node on a per–node basis and not per–processor as in the simulator. Also the time–stamps in the implementation are per node and not per processor, leading to less fine grain synchronization intervals.

Synchronization: The actual system, like the simulator, implements centralized barriers. The lock implementations in the simulator and the actual system are very similar.

Invalidation costs: In many applications, during a barrier synchronization operation processes are often required to invalidate a large number of pages. Since the cost of an invalidation is high (a single call to `mprotect` is about 10-15 μs), invalidating thousands of pages contributes noticeably to the SVM system time. The primary reason for this relatively high cost is the local TLB flush operation required. To reduce this cost we tried a few schemes and found a simple strategy to help in many of the applications. Contiguous pages are invalidated with a single call. However, even when there is a set of non–contiguous pages within a range, if more than a certain fraction of the pages within the range requires invalidations then we invalidate the entire range with a single call (when it is safe to do so). This helps to reduce barrier costs by up to 30% in applications such as LU contiguous and Water-spatial. Rough simulations of a few protocol alternatives show that the increase in page faults due to the unnecessary invalidations are quite small in all applications.

Protocol processing: In the implementation, protocol processing can be handled either using a process that occupies a processor and spins, or by having a sleeping process that is woken up by requests from the network. We rely on the scheduling in Linux for scheduling the protocol processes after it is awaken. We did not explore the version using a separate polling processor, since we expected the occupancy of this protocol processor to be quite low [49, 31]. The cost of protocol overhead can be observed to be low in almost all the application breakdowns we examined.

Protocol data structures layout: One interesting side–issue to our implementation was a surprising observation about the SMP memory subsystem performance within the node. Since protocol data structures are shared among all processes within a node, during some operations (especially barriers in well–balanced applications) there is a great deal of contention for certain data structures. We observed that in some scenarios, on the average, even a *single* reference to one of these structures required over 100 processor cycles, which is much higher than the unloaded memory access latency of about 20 cycles. An example of this is a vector used in the protocol page table used to indicate the state of memory protection for this page for the processors in the node. Components of this vector corresponding to different processes are written by those processes only but are next to one another in the vector. This causes a lot of false sharing. The solution is to arrange the state information by process rather than by page, placing the entries for a single process in an array laid out contiguously in memory. This is a fairly common problem and required a simple fix but it resulted in a surprising increase in performance.

5.3 Network Interface and Protocol extensions

In this section we describe the proposed extensions to the communication layer (network interface firmware and software) and the key protocol costs to which each mechanism is targeted. The resulting protocol, SVM-NI, removes some of the key bottlenecks in the system by taking advantage of these extensions.

Interrupt cost has been identified as a key bottleneck for SVM systems [12]. Reducing the number of interrupts is therefore one of the primary goals of this chapter. One way to avoid interrupts is to use polling on the main processor to handle messages. This approach has been investigated in Cashmere-2L [54] where the main processor polls the network queues for incoming messages on program back-edges. This requires that the code be instrumented to add polling instructions, and affects the time when messages are handled. A similar polling method was used and compared with interrupts in [101]. The tradeoffs between interrupts and polling are not very clear (the performance differences are found to be small for the platform used in [101]). Both methods use the main processor at the destination and both have disadvantages; interrupts rely on the operating system which makes performance (even when good) less predictable and portable, and polling introduces instrumentation costs and turns asynchronous events into synchronous ones by processing protocol requests at specific points during execution.

Our approach of using network interface support to reduce processor involvement eliminates interrupts, for both data and synchronization transfers, and at the same time improves performance further by reducing other protocol costs. It is different from both interrupts and polling in that the processor is not informed of when messages come in or when requests are serviced. Message handling is thus asynchronous with regard to the main processors, which means that traditional protocols that rely on the processors being informed of protocol activity must also be altered to ensure correctness. Protocols are further altered for performance by taking advantage of the available features of the communication architecture.

As mentioned, our base protocol, HLRC-SMP [9, 75], uses the communication layer only as a fast message passing system; no special features of the communication layer are used. The protocol uses four compute processes per–four–processor node, and an extra, floating process to handle protocol requests. Each incoming protocol request causes an interrupt that schedules the protocol process on one of the processors, as determined by the scheduler in the operating system. The protocol requests that require interrupts in the base protocol are page fetches, lock acquisition, and diff application at the home. We now describe the basic mechanisms we implement in the network interface to handle incoming requests or data transfers and hence reduce interrupts and asynchronous protocol processing. We assume a communication layer that supports the memory mapped communication model, present in many state of the art communication systems [23, 37]. In each case, we describe how the mechanisms are used to alter the protocol itself.

Remote writes to protocol data structures: We use a feature present in many state of the art communication layers, the ability to perform low overhead asynchronous send operations. Moreover, the memory mapped communication model we use already allows for data transferred to a remote node to be deposited in specified destination virtual addresses in main memory without involving a remote compute processor. Similarly, many communication systems support this or similar type of operations. We use this low–overhead, remote–deposit capability to directly update remote protocol data structures. Page and lock time–stamps are updated directly remotely without receiver processor intervention for protocol processing and write notices for multiple synchronization intervals are sent directly to the remote nodes, without being packed into bigger messages.

Page fetches: We extend the communication system to support (in firmware) a remote fetch operation to fetch data from a remote node. The remote fetch operation can fetch data from exported portions of the remote memory to arbitrary addresses in the local memory. We use this operation to fetch page data from the home. A node exports the pages for which it is the home, as well as protocol data. When a remote page must be requested, the local processor first requests the time–stamp or version number of the remote page and then requests the page itself. The request messages are made with asynchronous operations, so the request for the page is pipelined with the request for the page time–stamp. If the time–stamp is determined to be incorrect, the requester retries and requests the page time–stamp and the page itself anew. In practice this does not appear to happen very often, so no backoff scheme is currently employed to limit retries.

Remote diff application: Next, we use the low–overhead remote–deposit capability of the communication layer to remotely apply the diff and update shared data at the home nodes. When the local processor computes a diff, instead of storing it in a local data structure and then sending this diff data structure to the home of the page, it directly sends each contiguous run of different words to the home as it compares the page with its twin; this avoids the processing of the diff at the receiving node. We call this method of computing and applying the differences in shared data *direct diffs*. This saves the cost of packing the diff, interrupting a processor at the home of the page and having it unpack and apply the diff on the receive side. However, it may substantially increase the number of messages that are sent, since it introduces one message per–contiguous run of modifications within a page rather than one message per page that has been modified. Using a scatter–gather operation to remotely apply the modifications may combine some of the benefits of both worlds. However, a scatter–gather operation may require, depending on its semantics, substantial processing on behalf of the network interface. Thus, since the network interface we use employs a very slow processor, we do not explore the scatter–gather operation in this work.

We should note that direct diffs must be used in conjunction with the remote fetch mechanism for page fetches (as opposed to interrupting the home to request a page) as described earlier. The reason is that, since direct diffs do not require a home processor to be interrupted, the home processor cannot know when it has the updated version of a page so that it can service page requests that arrived earlier and have been waiting for that version. When remote fetches are used, each processor retries whenever it fails to fetch the right version of a page, and does not depend at all on the remote processors.

Lock synchronization: The algorithm that implements locks in the base HLRC-SMP protocol is used in many other protocols as well. Every lock is statically assigned a home. When a process needs to acquire a lock it sends a message to the home of the lock. The home forwards the message to the last owner and the owner releases the lock to the requester. The requests at the home and at the last owner are both handled using interrupts. The home of each lock is thus used to maintain a distributed list of locks. The last owner keeps the lock until another processor acquires the lock.

When the owner passes the lock along to another process it also sends along with the lock the invalidations that the requester needs to have to maintain the release–consistent view of the shared memory. Thus, since the base protocol uses lazy propagation of invalidations, when the protocol handler services a remote acquire it sends to the requester both the lock (mutual exclusion part) and the page invalidations (coherence information).

To avoid the asynchronous protocol processing and the interrupts, in the Final version we separate the mutual exclusion and the coherence information parts for all versions after the Base version. For the coherence part, we propagate invalidations to other nodes in the system eagerly at releases, using the remote deposit capability of the communication system. Invalidations are still applied lazily at the next causally related acquire. Another way to decouple coherence information and mutual exclusion, without using interrupts and still maintaining lazy invalidation propagation, is to use the remote fetch mechanism described above, to fetch write notices at acquires rather than propagate them eagerly at releases. However, there is a tradeoff here between the acquire latency and the number of messages, and we decided to stay with an eager propagation and lazy application approach for write notices.

For the mutual exclusion part, we extend the communication layer to provide the programmer with network locks. Lock acquisition and release for mutual exclusion become communication system, rather than HLRC protocol operations. The implementation of the locks, in the network interface firmware, is similar to the algorithm described earlier. The network interface maintains a table of locks. For each lock it stores one word of state and ownership information, and a time–stamp. The network interface does not need to manipulate the time–stamp; it is a piece of information related with each lock. Thus, no coherence information is involved in the manipulation of locks from the network interface. The table that keeps the information for the locks in the communication layer is currently kept in the network interface (a few tens of KBytes per node). However, if necessary it can be paged to host memory.

Essentially, after decoupling the coherency and mutual exclusion parts for synchronization in the protocol, instead of using host processors in the nodes to maintain the distributed lists for locks using interrupts, the network interface processors do so without host processor involvement.

Summary: The final version of the protocol (SVM-NI) that takes advantage of all these extensions does not use interrupts or polling at all, either for protocol data, or for application data, or for synchronization. The new model is that, instead of getting access to remote resources through interrupt handlers, processors do so by using standard remote operations implemented in the network interface. These operations are general purpose operations that extend the functionality of the network interface, and their implementation does not require receive host–processor intervention.

5.4 Experimental Testbed

To demonstrate the benefits of our approach, we implement HLRC-SMP and the network interface extensions on a cluster of Intel Pentium Pro SMPs connected with Myrinet. The nodes in the system are 4–way Pentium Pro SMPs at 200 MHz. The Pentium Pro processor has an 8 KByte data and an 8 KByte instruction cache. The processors are equipped with a 512 KBytes unified 4–way set associative L2 cache and 256 MBytes of main memory per node. The SMP nodes run the SMP version of the Linux operating system (version 2.0.24). Table 5.1 shows the cost of some basic system operations.

Operation	Cost (μs)
`mprotect`	12
Read page fault (null)	20
Write page fault (null)	20
Local memory `bcopy` (4 KBytes)	150
one–way VMMC latency (4 Bytes)	~19

Table 5.1: Basic system costs. All costs are in μs.

Myrinet [17] is a high–speed system–area network. A Myrinet network is composed of point–to–point links that connect hosts and switches. Each network interface in our system has a 33 MHz programmable processor, 1 MByte of SRAM, and connects the node to the network with two unidirectional links of 160 MByte/s peak bandwidth each. Actual node–to–network bandwidth is constrained by the 133 MBytes/s I/O bus on which the NI sits. All four nodes are connected directly to an 8–way switch.

The communication layer that we use on top of the Myrinet network is VMMC [23]. VMMC provides protected, reliable, low–latency, high–bandwidth user–level communication. On a network of SMPs, the one–way VMMC latency is about 19μs for one–word messages and the maximum VMMC bandwidth about 95 MBytes/s (see Chapter 2). The basic feature of VMMC that we use in this work is the remote deposit capability. The sender can directly deposit data to exported regions of the receiver's memory, without receiving side (process or processor) intervention. We extend VMMC with remote fetch and lock support as discussed earlier. The time for a one–page remote fetch operation is about 110μs (about 40μs for one word).

The communication layer of the system includes a performance monitoring tool [60] that gathers network–level data. The core of the monitor is firmware run by the NI processor on the Myrinet NI card. Working at the NI firmware level gives us access to crucial determinants of application performance not available in higher–level software. Several pieces of the tool's functionality would be infeasible or costly to implement in higher–level software, such as a global synchronized clock and the ability to measure network–level rather than end–to–end latencies.

5.5 Performance Results

The measured costs of various HLRC-SMP basic protocol operations in our environment are shown in Table 5.2.

The cost of the page–fetch operation is much faster than previous implementations of SVM systems; for instance, TreadMarks [52] reports over 1900 μs (for a 4 KBytes page) and SoftFLASH [29] states 1164μs (for a 16 KBytes page at the Kernel level on an SGI Challenge). Cashmere–2L [91] reports the page fetch operation to cost about

Micro–benchmark	Cost (μs)
Page transfer b/w	50 MBytes/s
Page bcopy	26 MBytes/s
Page fetch	200
Remote Lock Acquire	220
Barrier (16 procs)	500

Table 5.2: Basic HLRC-SMP costs. All costs are in μs, except if otherwise noted. The Lock Acquire and the Barrier operations do not include the cost of exchanging coherency information (invalidations). The lock acquisition cost is approximate since it is averaged over many acquires.

800 μs.

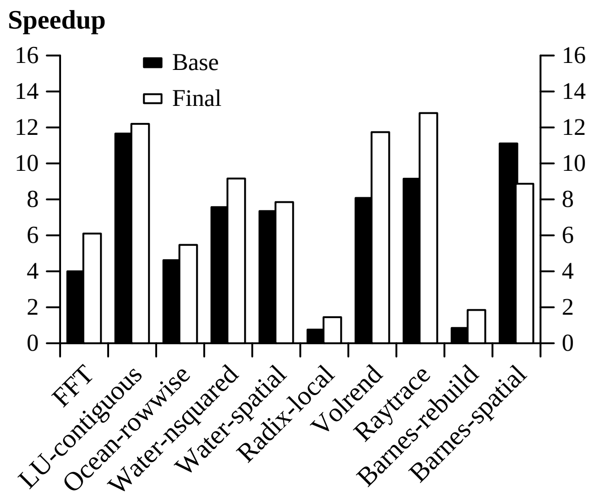

Figure 5.2: Application speedups. The left bar is the base protocol and the right bar the protocol with the NI extensions.

In the next paragraphs we evaluate the new protocols that result from our extensions to the network interface. In the presentation of the results we evaluate the features and resulting protocols cumulatively in the order in which they were discussed. Since one of our goals is to see how well the final SVM systems with all extensions can perform relative to hardware coherence, we use versions of the SPLASH-2 applications that are restructured to perform much better on SVM systems where relevant [47]. (For Barnes we present both the original and the restructured version.) The problem

Application	Problem Size	Uniproc Time	Base spdup	Final spdup	Overall improv.	Data Wait Time improv.	Lock Time improv.
		(sec)			(%)	(%)	(%)
FFT	4M points	4.6	4.0	6.1	52.50	45.37 (44.92)	0.00
LU-cont	4096x4096 matrix	935.9	8.77	12.20	4.63	13.46 (11.20)	0.00
Ocean-row	514x514 ocean	248.3	2.41	5.47	18.40	21.76 (19.26)	9.21
Water-nsq	4096 molecules	360.6	7.57	9.16	21.00	15.26 (46.17)	62.76
Water-spa	15625 molecules	157.2	6.12	7.85	6.80	41.60 (41.80)	9.69
Radix-loc	4M keys	5.9	0.76	1.45	90.79	26.76 (27.00)	53.21
Volrend-stl	256^3 cst head	13.2	9.15	11.74	45.30	43.81 (42.44)	50.44
Raytrace	car	29.8	8.08	12.8	39.89	2.52 (50.03)	59.01
Barnes-reb	32K particles	47.7	0.85	1.85	117.65	41.07 (68.25)	1.98
Barnes-spa	128K particles	219.2	11.11	8.87	-20.16	40.99 (37.70)	33.84

Table 5.3: Application statistics. The sixth column represents the overall, percentage improvement in each application between the Base and Final systems. The seventh column is the percentage improvement in data wait time between DW and DW+RF and the eighth column, the percentage improvement for lock time between DW+RF+DD and Final. For remote fetch we also report the percentage improvement between DW and Final in parentheses.

size we choose are large enough to avoid artifacts and close close to the sizes of real world problems. Table 5.3 presents the problem sizes along with the uniprocessor execution times (which are often small because we run only a few time-steps or frames of an application). Speedups are computed between the sequential program version (without linking to the SVM library or introducing any other overheads) and the parallel version. The initialization and cold-start phases are excluded from both the sequential and the parallel execution times in accordance with SPLASH-2 guidelines.

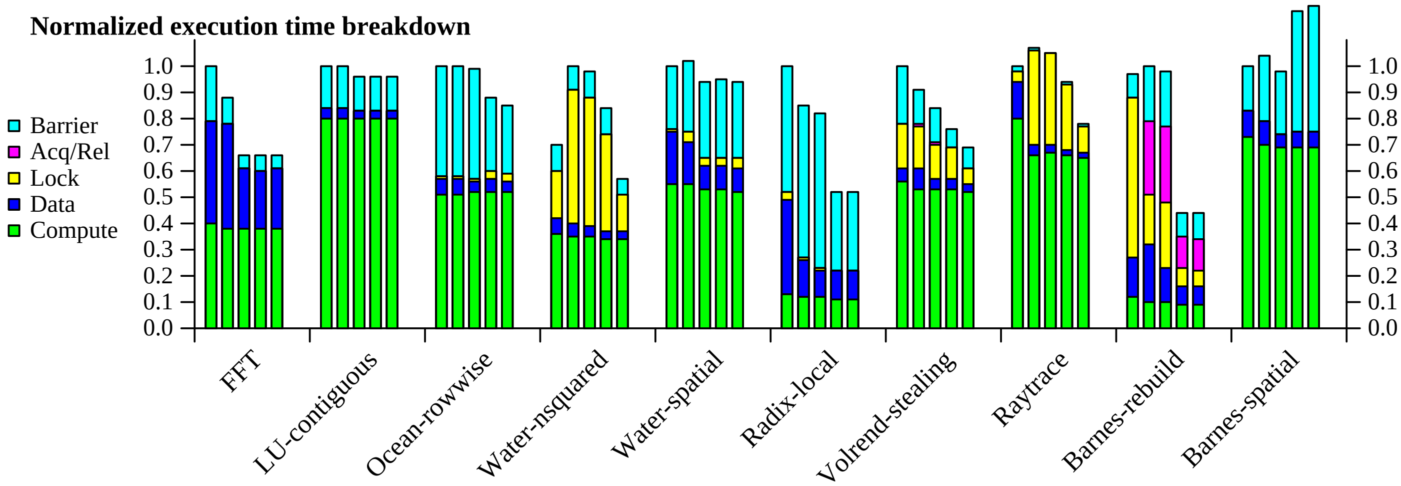

Figure 5.3: Normalized execution time breakdowns. From left to right the bars for each application are (i) HLRC-SMP (Base), (ii) direct deposit (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

Figure 5.2 shows the speedups for the base and final protocols and Figure 5.3 the average execution time breakdowns. Table 5.3 presents in numbers the information given by Figure 5.3. Figures 5.5–5.42 [1] show the per–processor execution–time breakdowns for selected application for each version of the protocol. The protocols we evaluate are:

Base: This is the base protocol HLRC-SMP, where VMMC is used only as a fast communication layer. The speedups for this case are shown in Table 5.3 and in Figure 5.1. From the analytic execution time breakdowns we see that different applications exhibit different bottlenecks. This suggests that multiple aspects of the protocol need to be attacked for overall application performance to be improved. In FFT the execution time is divided between compute, data wait, and barrier time, with the first two components being more significant. We see that improving data wait is what can give the highest benefit. In Ocean-rowwise the most important component

[1] We omit the analytic breakdowns for applications that either perform well with a protocols, or exhibit no significant change from analytic breakdowns shown for other protocols.

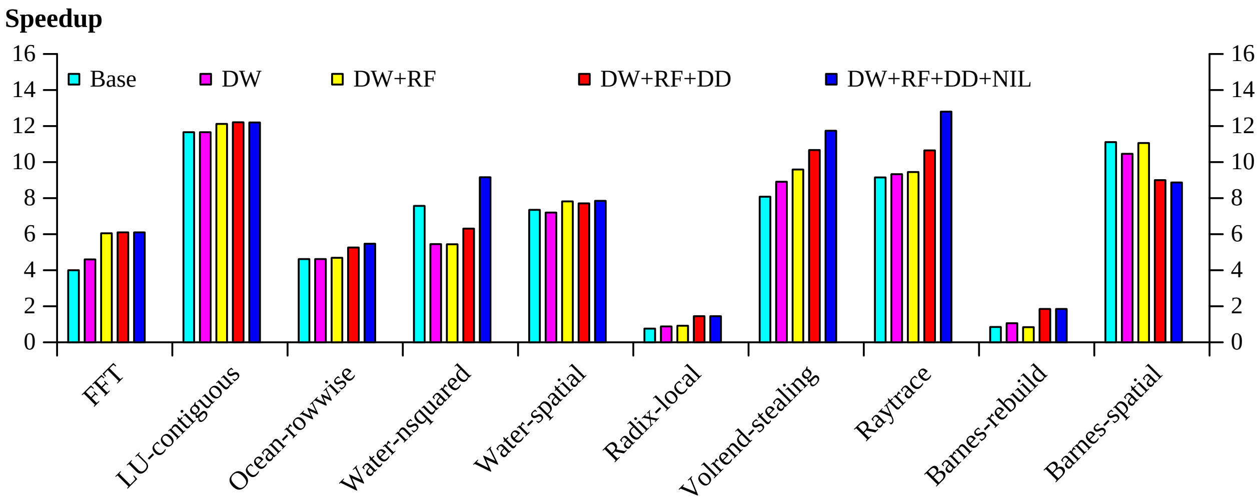

Figure 5.4: Application speedups on 16 processors. From left to right the bars for each application are (i) HLRC-SMP (Base), (ii) direct deposit (DW), (iii) remote fetch (RF), (iv) direct diffs (DD), and (v) network interface locks (NIL).

is barrier time. This component of the execution time is high due to the large number of barriers. In Barnes-rebuild and Water-nsquared the main bottleneck is lock time. LU-contiguous, Barnes-spatial and Raytrace perform relatively well even in the Base version. In Radix-local the main cost is the barrier time; in this case however, the high barrier time is due to expensive individual barriers rather than the large number of barriers. In Volrend the protocol overhead is divided among data wait time, lock

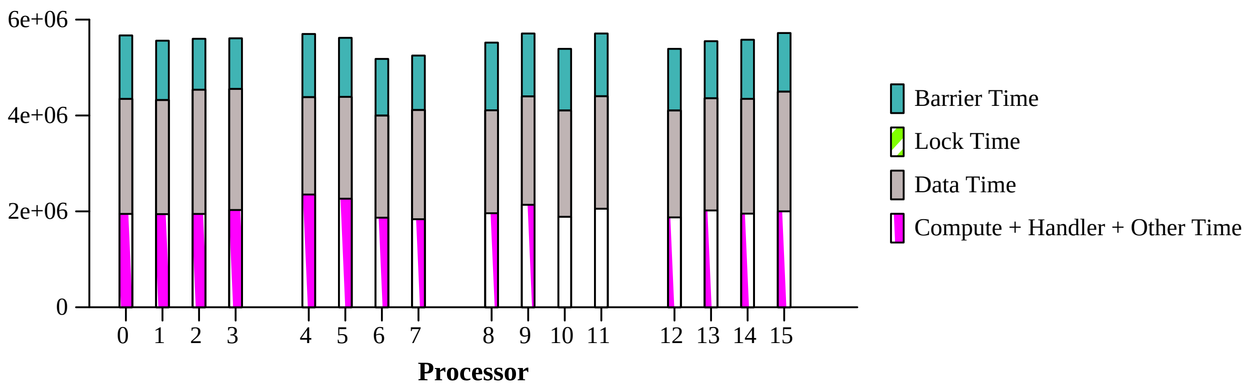

Figure 5.5: Execution time breakdown for FFT, Base.

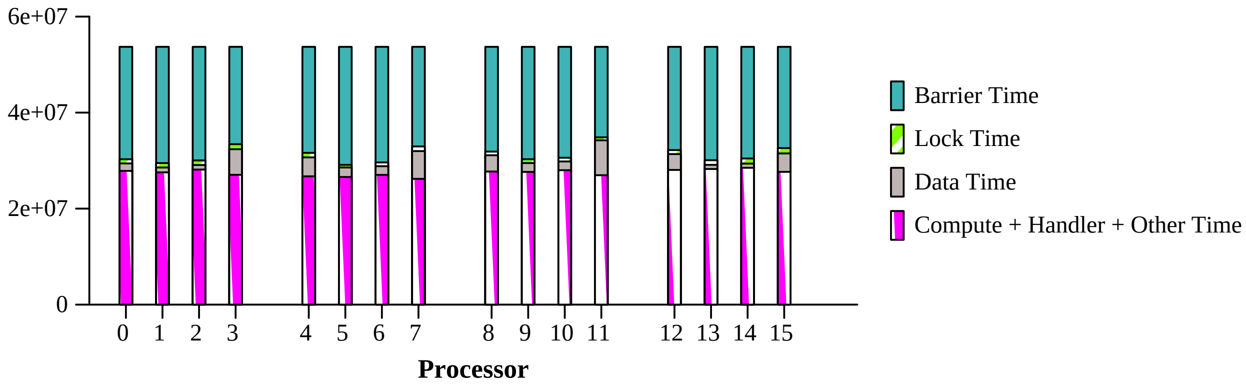

Figure 5.6: Execution time breakdown for Ocean-rowise, Base.

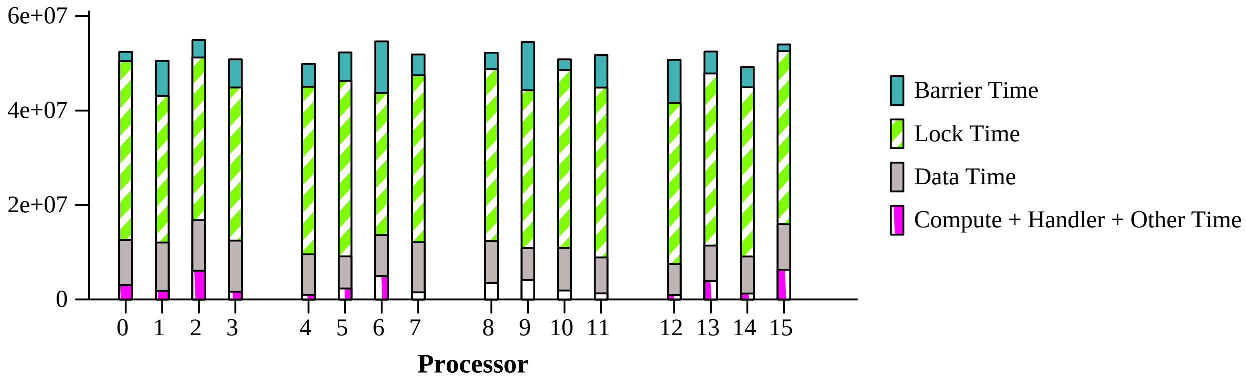

Figure 5.7: Execution time breakdown for Barnes-rebuild, Base.

time, and barrier time, whereas in Water-spatial it is divided among data wait time and barrier time.

Direct deposit (DW): This protocol uses the remote deposit capability of VMMC to directly update only remote protocol data structures (time–stamps and write notices). Shared data are propagated to the home of each page with the traditional diff mechanism, using interrupts at the home to apply the diffs. Since the overhead of the asynchronous send operation in VMMC is relatively small (about 1.5μs), this approach results in more pipelining: Instead of having the host processor pack data

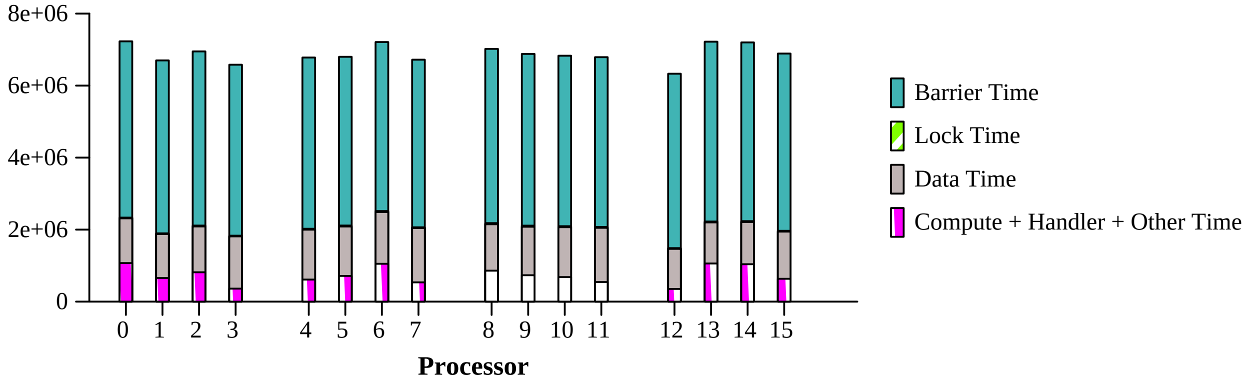

Figure 5.8: Execution time breakdown for Radix-local, Base.

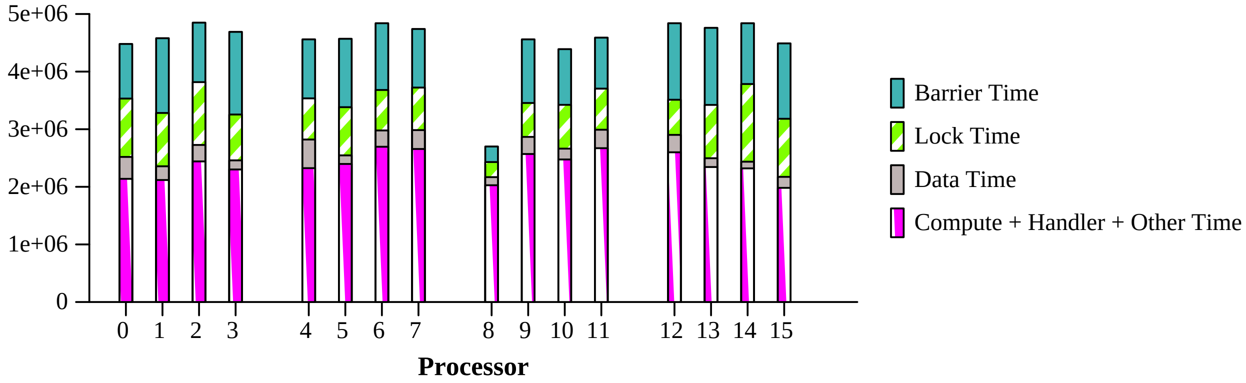

Figure 5.9: Execution time breakdown for Volrend, Base.

in one big message before the NI can take over, the host's posting of small messages using asynchronous sends is pipelined with the NIs preparation and sending of them. Invalidations are thus propagated eagerly, and synchronization is separated from coherency information.

We see that all applications with the exception of Water-nsquared perform either comparably or better with this DW than with the base protocol [2]. This is primarily

[2]Barnes-rebuild uses locks for flag synchronization with the Base protocol and acquire/release calls for the rest of the protocols. In the Base protocol it is very difficult to implement the acquire/release primitives efficiently, because mutual exclusion and control information propagation are bound together. The result of this difference is that the number of lock operations and the order in which they are called differ between the Base and the DW protocol. However, as we see from the breakdowns the difference in the execution time between these two protocols is not affected

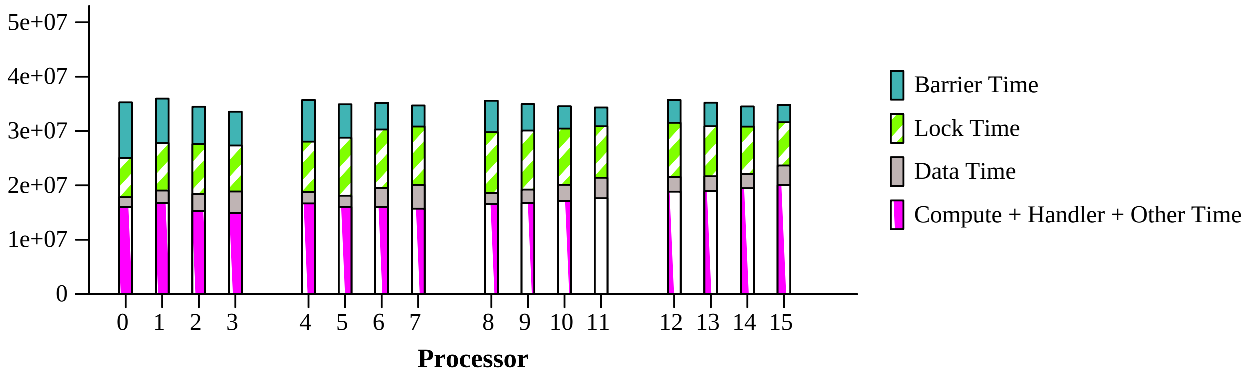

Figure 5.10: Execution time breakdown for Water-nsquared, Base.

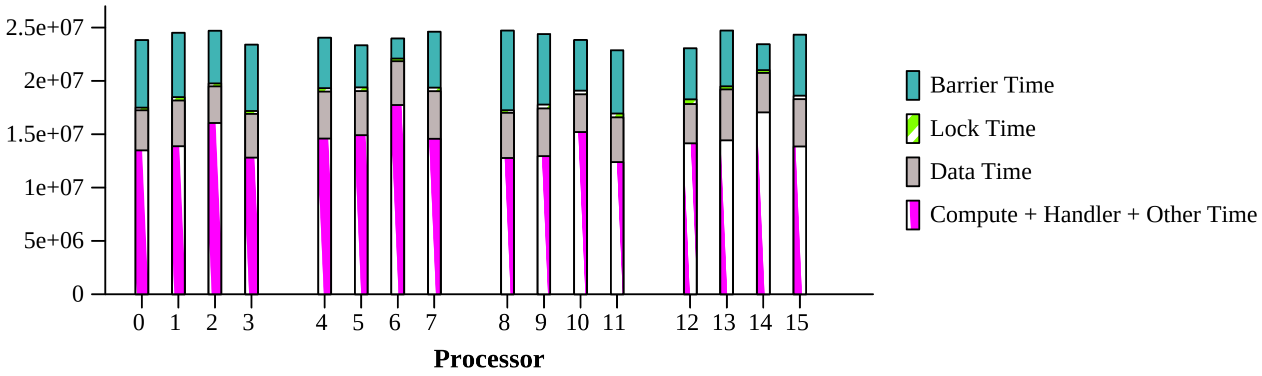

Figure 5.11: Execution time breakdown for Water-spatial, Base.

due to the removal of protocol message processing at the receive side that now use direct deposit. However, the DW protocol does send more messages, both because it uses eager propagation and because it uses small messages. This is in fact the reason that Water-nsquared performs worse. This version of Water-nsquared uses fine–grained, per–molecule locks when updating the private forces computed by each process into the shared force array, to reduce inherent serialization at locks. However, this causes the frequency of locks and hence of invalidation propagation to be very large. Thus, even if each lock release operation sends a small number of messages

dramatically by this difference, as expected, since the protocol overhead is similar in both cases.

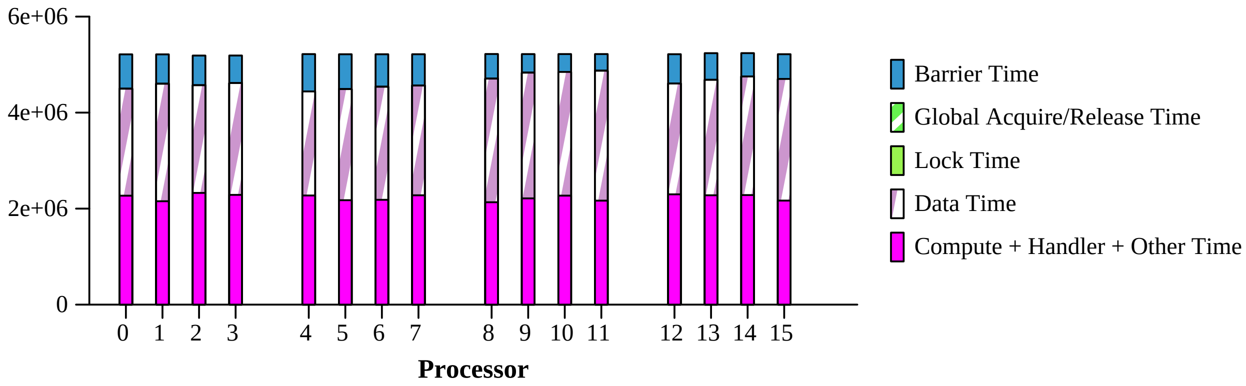

Figure 5.12: Execution time breakdown for FFT, DW.

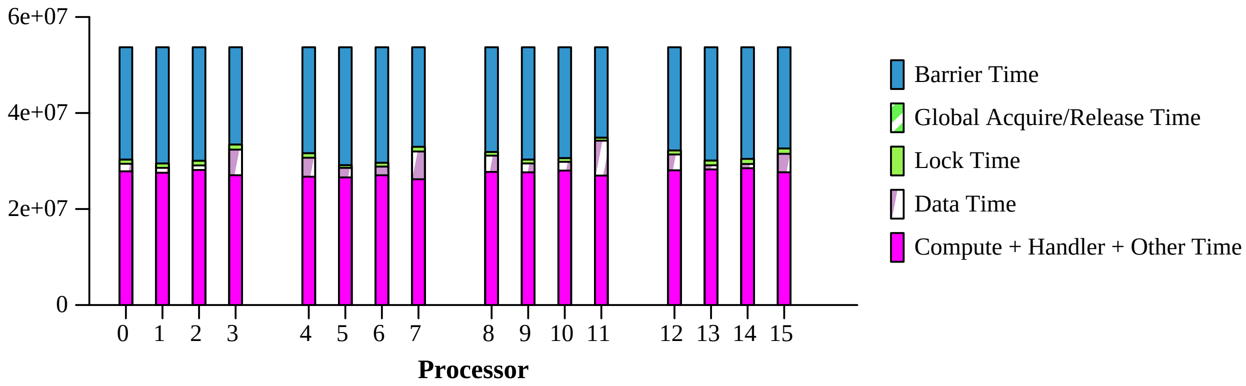

Figure 5.13: Execution time breakdown for Ocean-rowise, DW.

(which is true for Water-nsquared, where one message per release is sent), the total number of messages is still increased, because of the eager nature of releases. The performance monitor shows that lock acquire messages are delivered in this case to the host with high delays. This means that the increased messages occupy the queues in the NIs, increasing the time it takes for a lock acquire message to get through and get a reply. It is worth noting here that in the Final version of the protocol the effect of this traffic problem is much less apparent and performance of Water-nsquared improves by 21% compared to Base. This happens because in the Final version lock messages need not be delivered to host memory but are handled completely in the

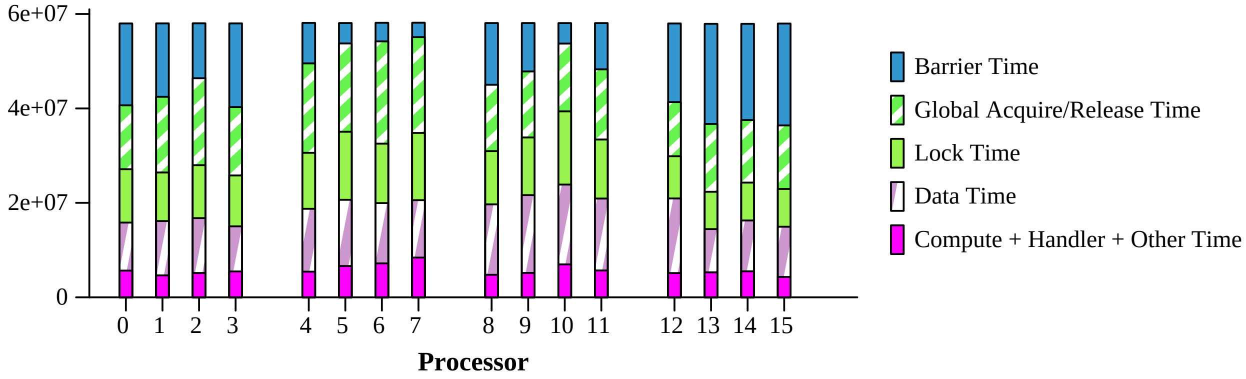

Figure 5.14: Execution time breakdown for Barnes-rebuild, DW.

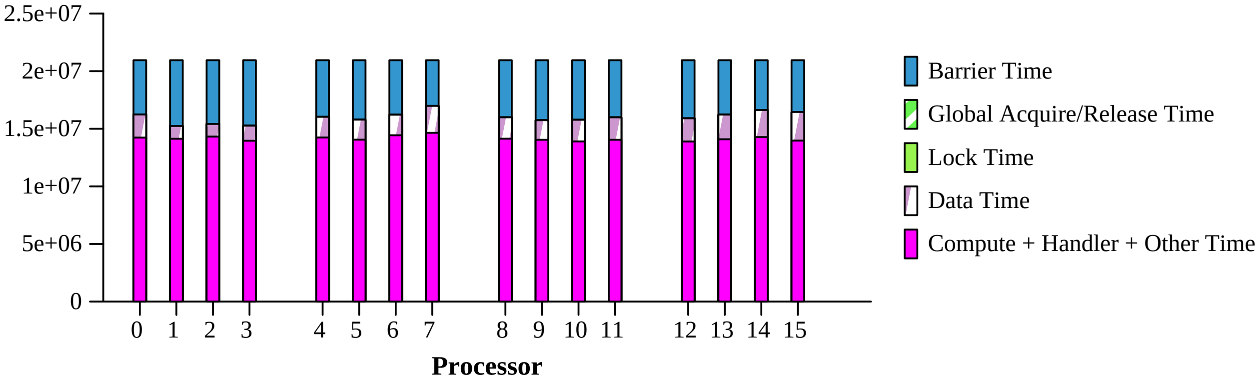

Figure 5.15: Execution time breakdown for Barnes-spatial, DW.

network interface. Thus instead of adding lock acquire messages to the queue of messages that are to be delivered to the host, where they would have to wait for all previous messages to be delivered, they are processed as soon as they are received, and the reply to the requester is sent directly by the network interface.

One way to fix the problem of increased messages introduced by the eager releases, is to propagate invalidations lazily at acquires using a scatter–gather operation to both perform direct deposits of the write notices from different intervals and reduce the number of messages. But this approach requires that interrupts be used to inform the last releaser when it needs to send the invalidations. Another approach is to

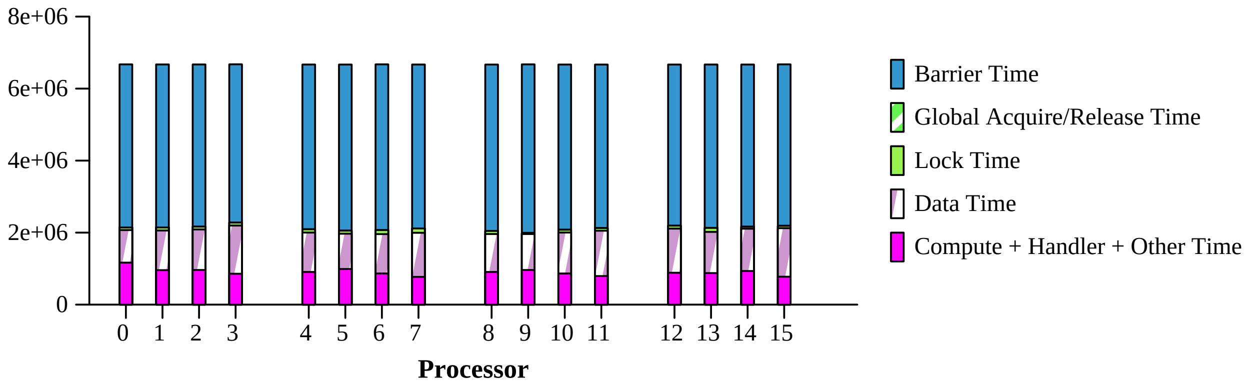

Figure 5.16: Execution time breakdown for Radix-local, DW.

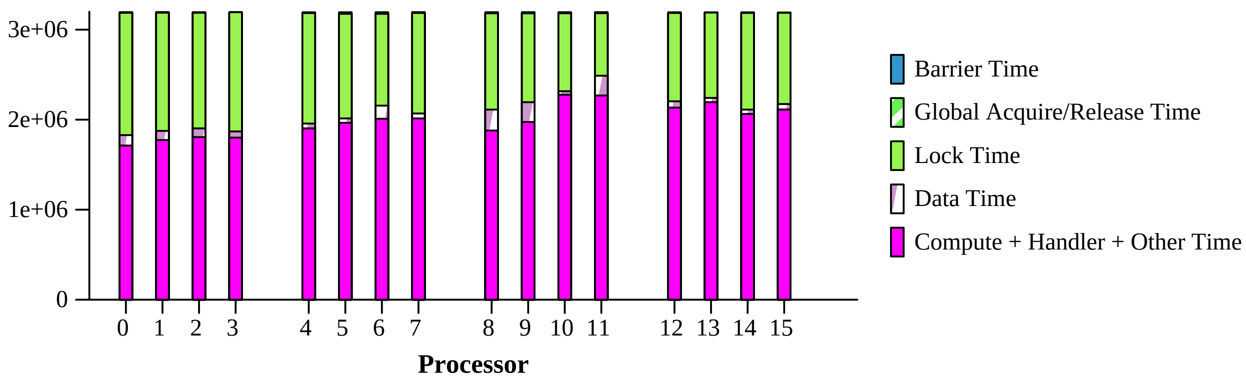

Figure 5.17: Execution time breakdown for Raytrace, DW.

implement lazy invalidation propagation in the protocol by taking advantage of the remote fetch operation and to change the protocol data structures such that one read can fetch the invalidation for multiple intervals (thus having fewer, larger messages). In this case the processor that acquires the lock will use the remote fetch operation to get the necessary invalidations from the last owner of the lock, without having to interrupt it. Both of these methods, however, may hurt the performance of other applications, since the latency of various operations, e.g. getting write notices, is incresed. It is not clear if there is one best method, and some of these tradeoffs are worth exploring.

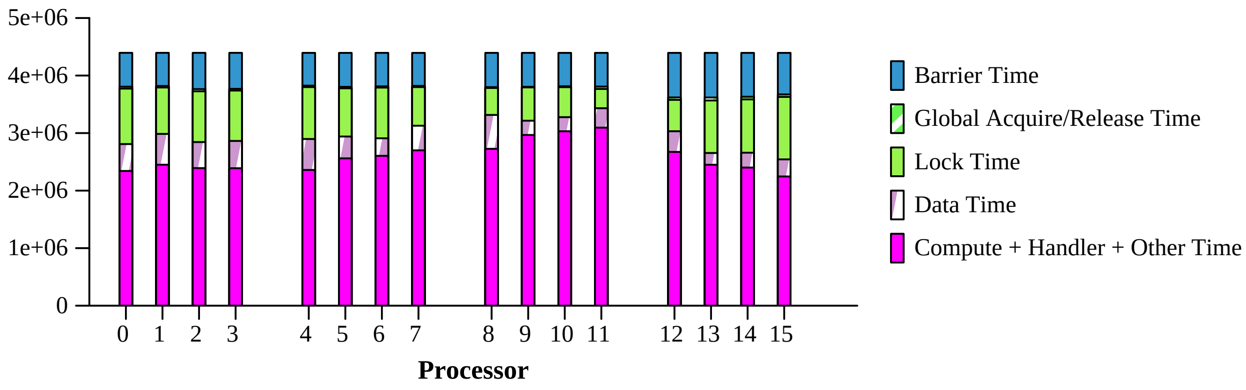

Figure 5.18: Execution time breakdown for Volrend, DW.

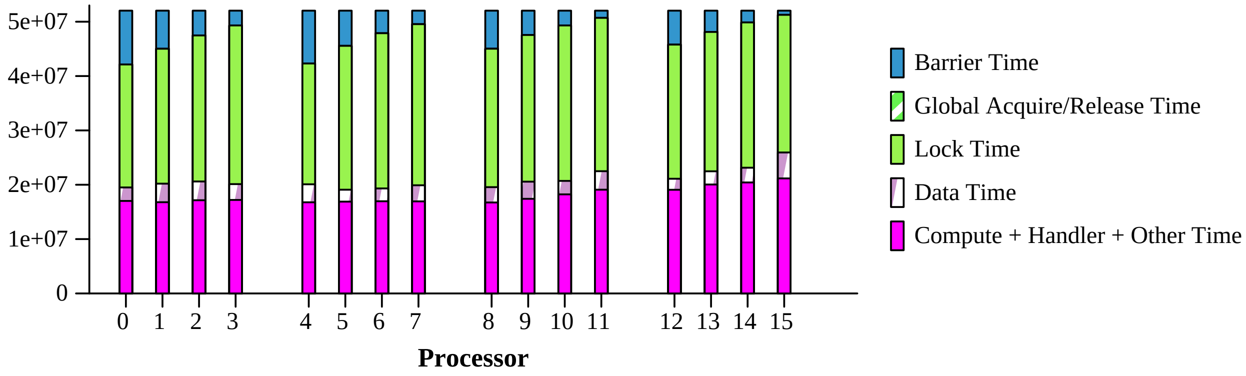

Figure 5.19: Execution time breakdown for Water-nsquared, DW.

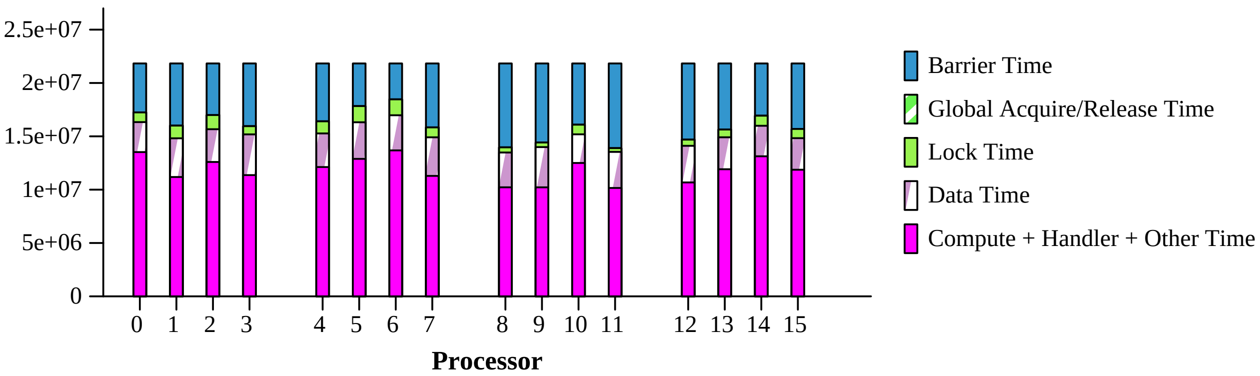

Figure 5.20: Execution time breakdown for Water-spatial, DW.

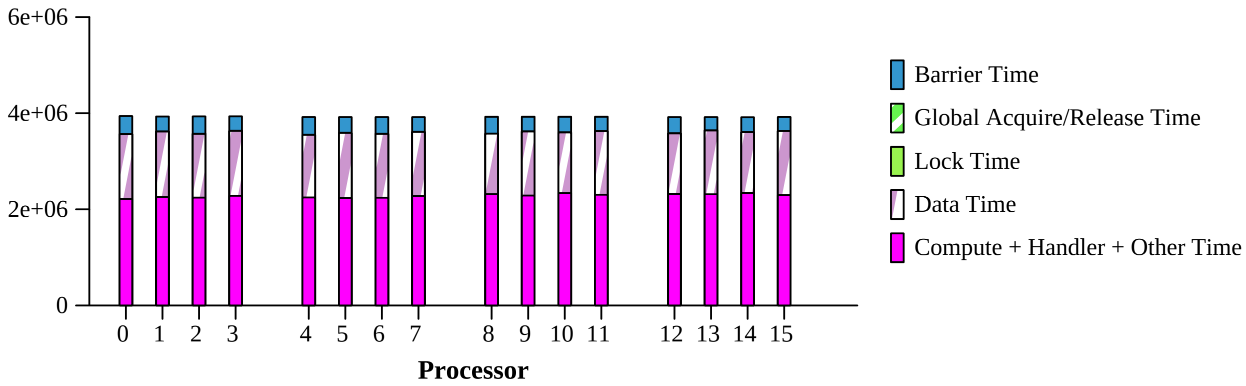

Figure 5.21: Execution time breakdown for FFT, DW+RF.

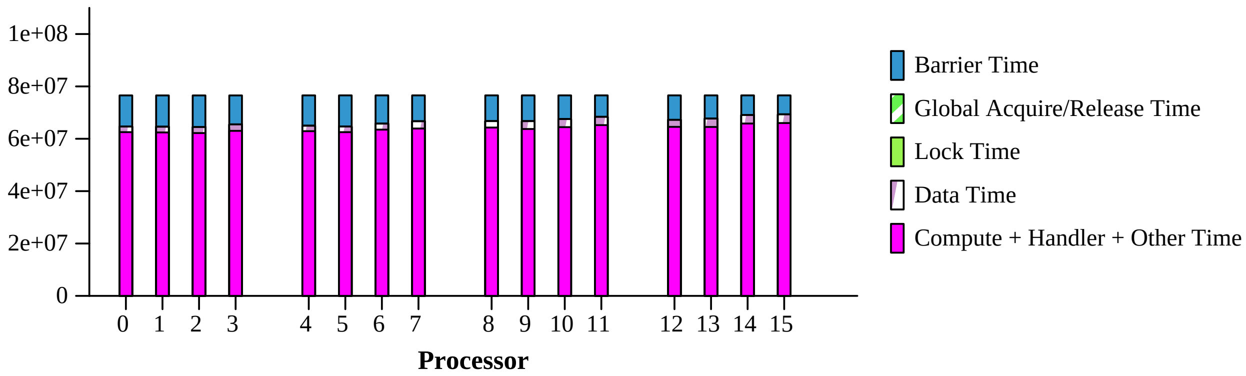

Figure 5.22: Execution time breakdown for LU-contiguous, DW+RF.

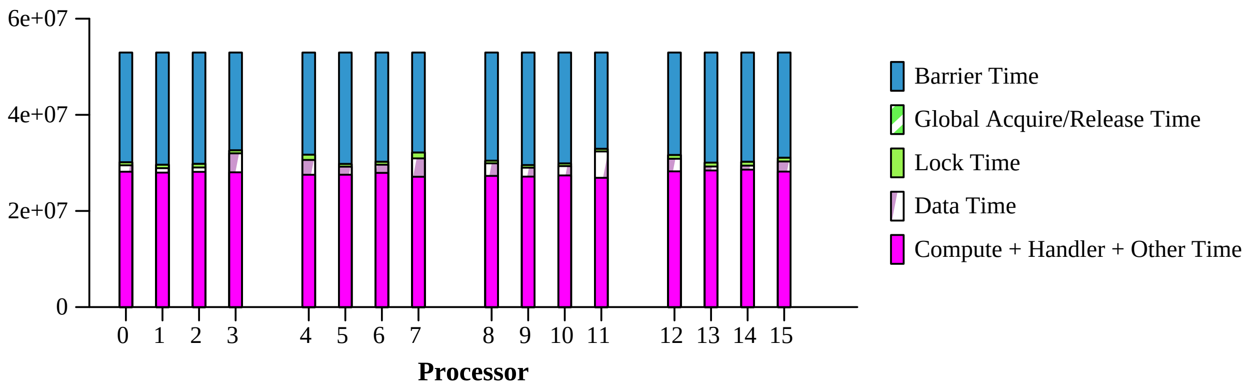

Figure 5.23: Execution time breakdown for Ocean-rowise, DW+RF.

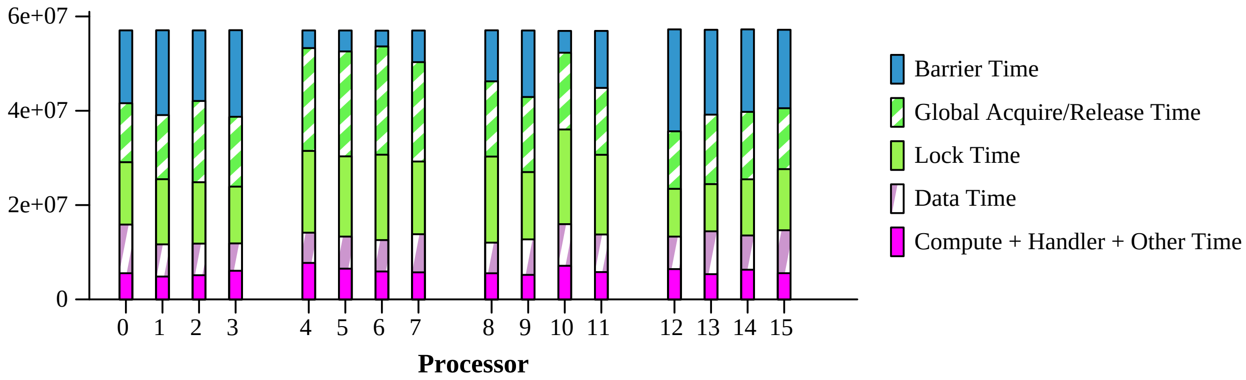

Figure 5.24: Execution time breakdown for Barnes-rebuild, DW+RF.

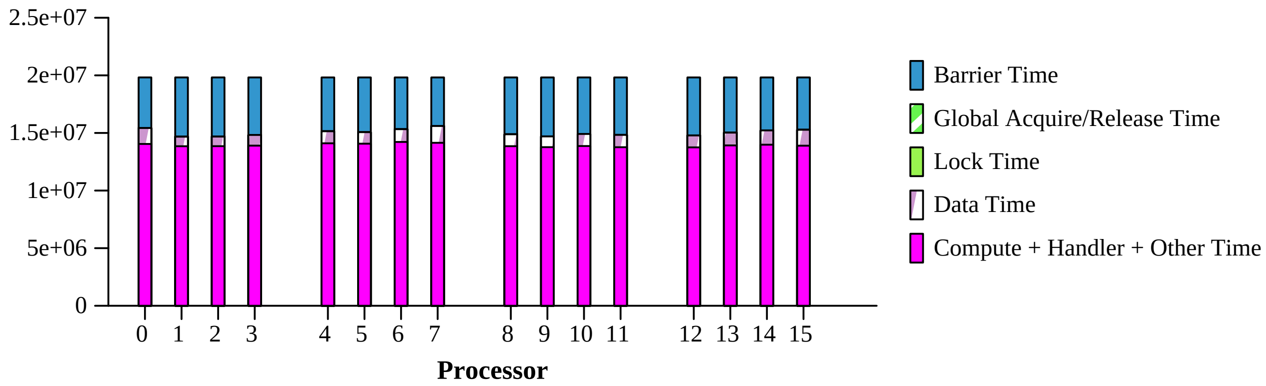

Figure 5.25: Execution time breakdown for Barnes-spatial, DW+RF.

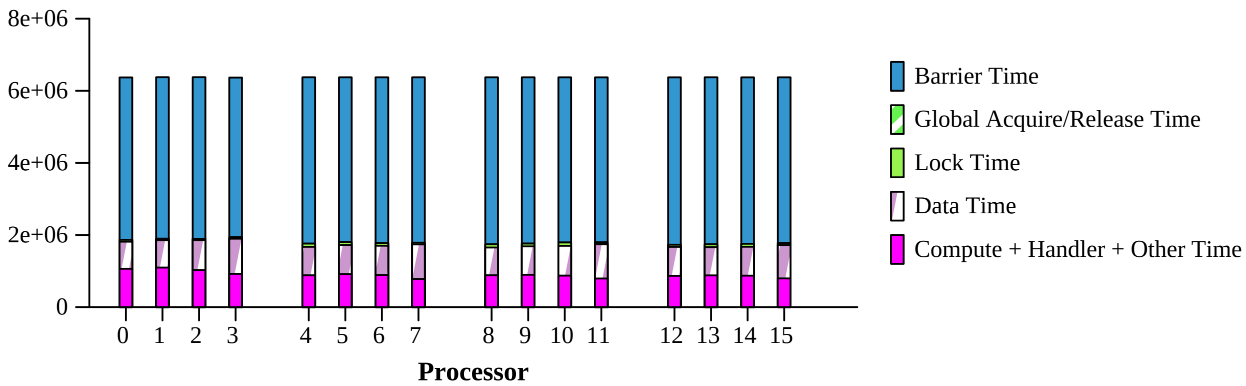

Figure 5.26: Execution time breakdown for Radix-local, DW+RF.

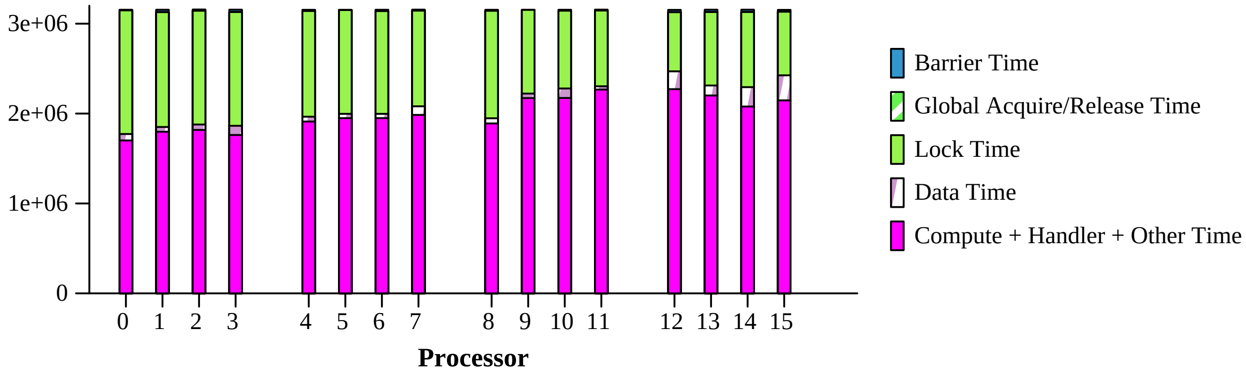

Figure 5.27: Execution time breakdown for Raytrace, DW+RF.

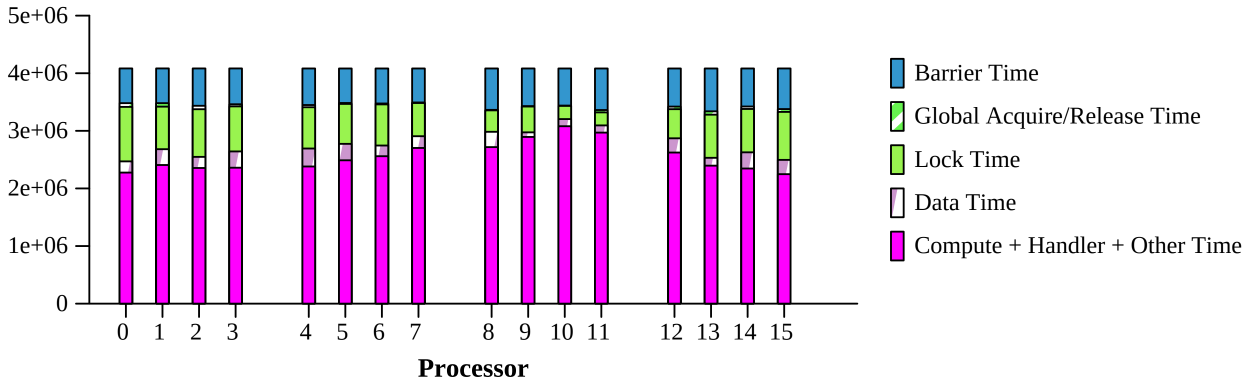

Figure 5.28: Execution time breakdown for Volrend, DW+RF.

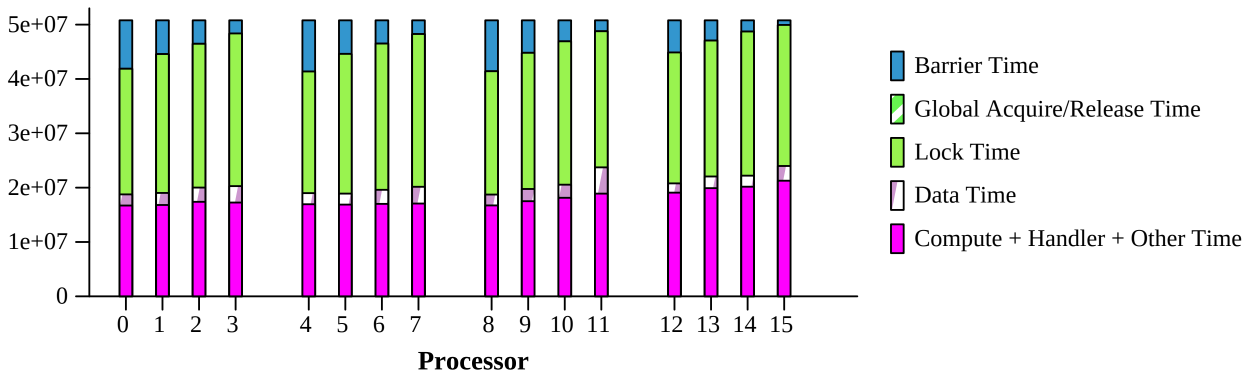

Figure 5.29: Execution time breakdown for Water-nsquared, DW+RF.

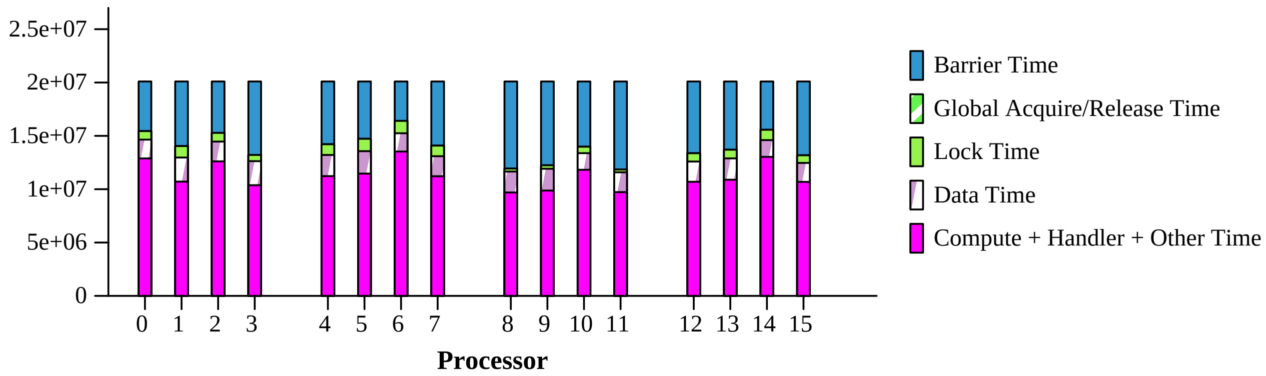

Figure 5.30: Execution time breakdown for Water-spatial, DW+RF.

Remote fetch (RF): In this protocol we introduce the remote fetch operation to fetch application and protocol data without interrupting the remote processor. The remote fetch operation is mainly used in this protocol to fetch pages and page time–stamps. Other uses (for implementing different algorithms for mutual exclusion or different levels of laziness in the protocol as mentioned in the previous paragraph) are also possible depending on the specifics of the protocol used. We see that all applications benefit from the use of remote fetch even beyond DW, to varying degrees. Applications with high data wait times, like FFT, Water-spatial, Radix, and Barnes-rebuild, see a high improvement in performance. The data wait time is reduced up to 45%, with most applications seeing improvements in data wait time of more than 20%. It is interesting to note that the improvement in data wait time in FFT (45%) comes exclusively from eliminating the interrupts and the related scheduling effects within an SMP. The use of the remote fetch operation does not reduce, in this case, the contention in the network. This extension to the network interface has fairly predictable results; the major protocol overhead reduced is the data wait time.

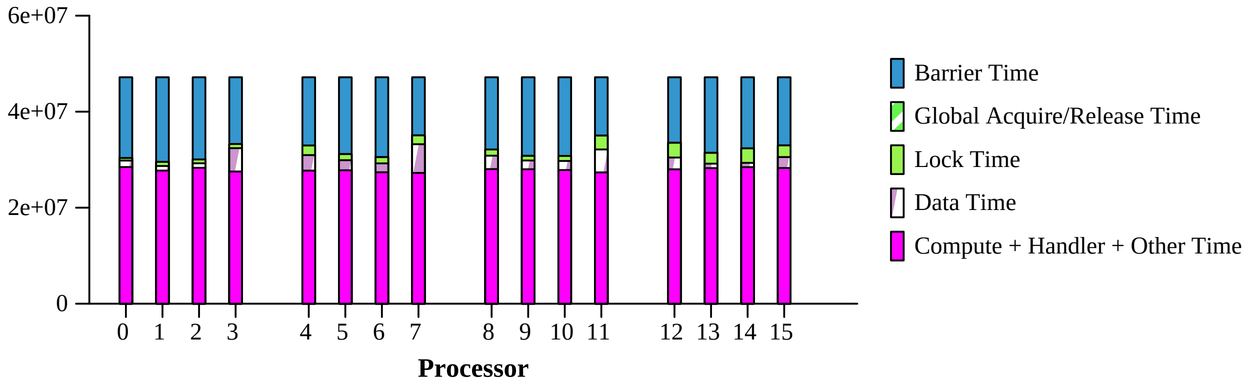

Figure 5.31: Execution time breakdown for Ocean-rowise, DW+RF+DD.

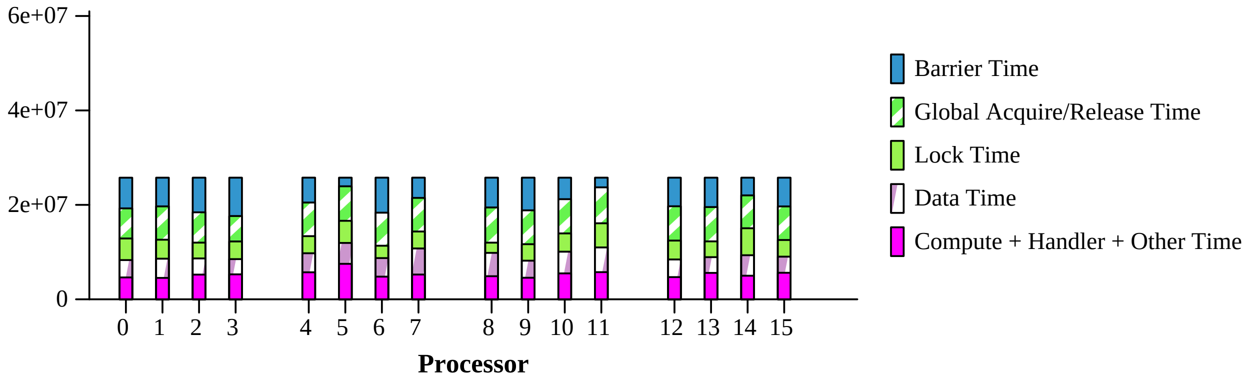

Figure 5.32: Execution time breakdown for Barnes-rebuild, DW+RF+DD.

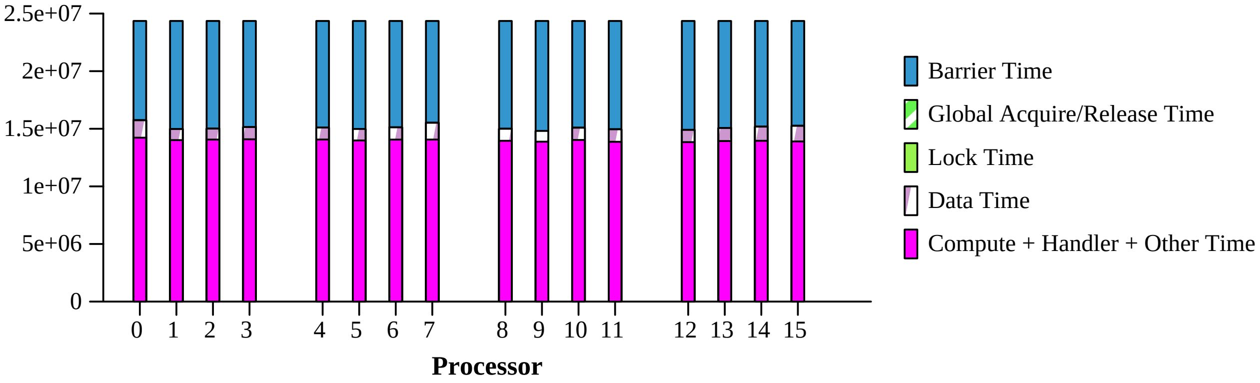

Figure 5.33: Execution time breakdown for Barnes-spatial, DW+RF+DD.

Direct diffs (DD): This protocol uses the remote deposit operation to update home pages and eliminates the need for interrupts in diff application. As we see from the execution time breakdowns in Figure 5.3, direct diffs are particularly useful in the irregular applications that have a lot of synchronization and hence diffs: Radix, Barnes-rebuild, Raytrace, Volrend and Water-nsquared. The benefits in performance come from eliminating the interrupts (and related scheduling effects in the SMP nodes) and from better load balancing of protocol costs, and they come despite the fact the direct diffs make messages even smaller (one per–contiguous run of modifications in a page). Especially for Barnes-rebuild and Radix-local, the reduction

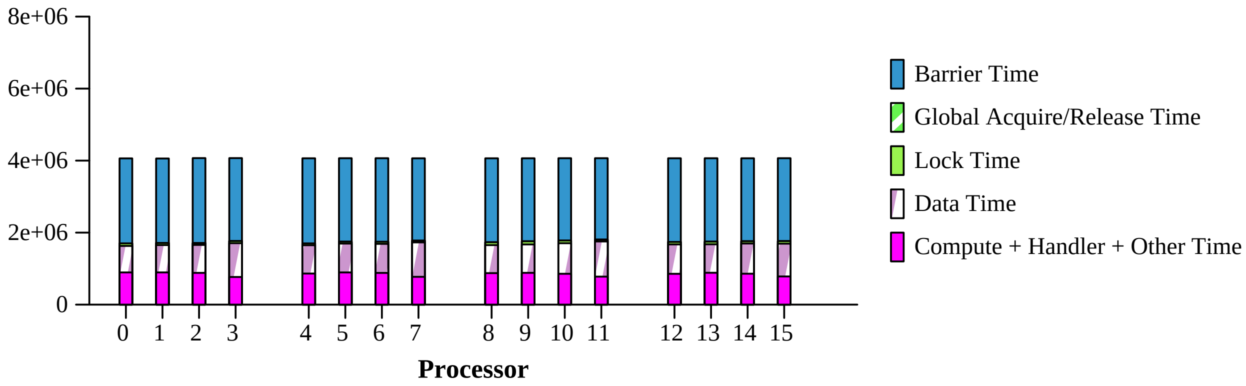

Figure 5.34: Execution time breakdown for Radix-local, DW+RF+DD.

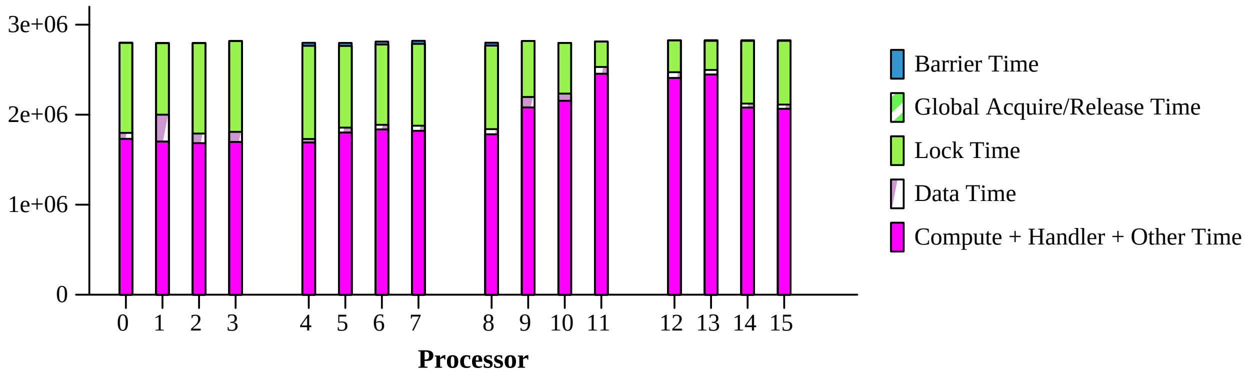

Figure 5.35: Execution time breakdown for Raytrace, DW+RF+DD.

of all protocol overheads is very significant. These applications exhibit a lot of synchronization; eliminating interrupts and scheduling effects at diff application leads to significant improvements in performance.

We also note that Barnes-spatial performs worse with DD. This is because the number of messages in the network increases by more than a factor of 30 due to the highly scattered nature of diffs within each page. This problem can be addressed either by changing the layout of data structures in the application or by adding a scatter–gather operation in the NI. Modifying the application such that writes to shared data by each processor are much more contiguous within each page, would re-

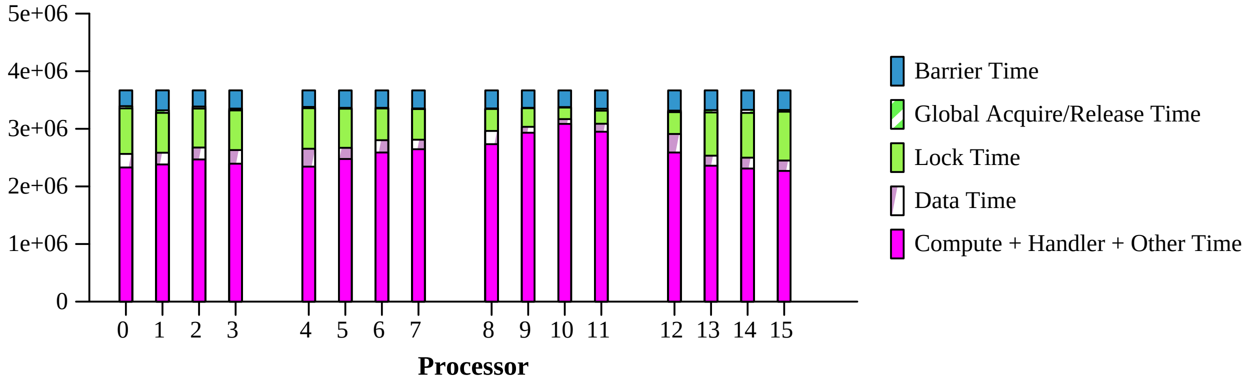

Figure 5.36: Execution time breakdown for Volrend, DW+RF+DD.

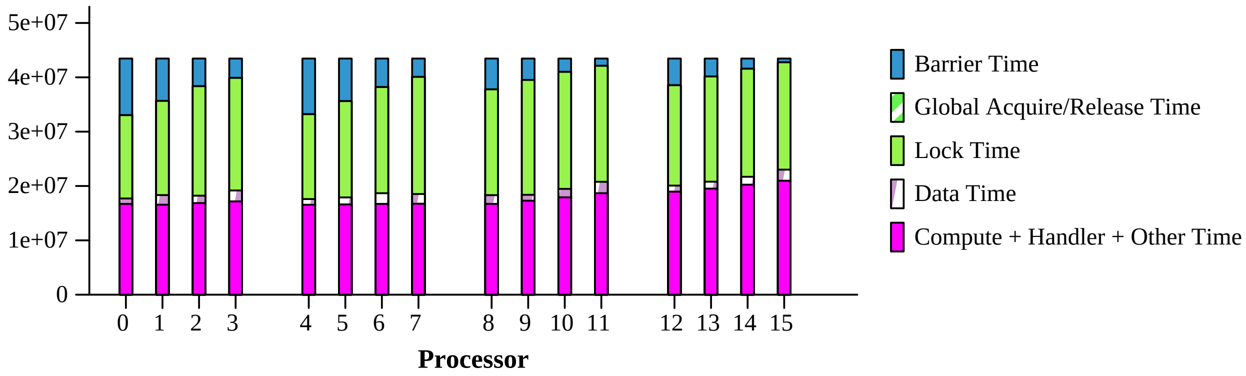

Figure 5.37: Execution time breakdown for Water-nsquared, DW+RF+DD.

sult in a smaller number of messages during diff computation. However, this change may require extensive changes to the application code. Using a scatter–gather operation combines some of the best features of both direct diffs and the traditional diff propagation mechanism since it minimizes both the number of messages and does not need to interrupt the receiving node. Note that this effect for Barnes-spatial disappears in all protocols when direct diffs are not used. Thus, using any combination of the network interface extensions does not decrease performance, as long as direct diffs are not used.

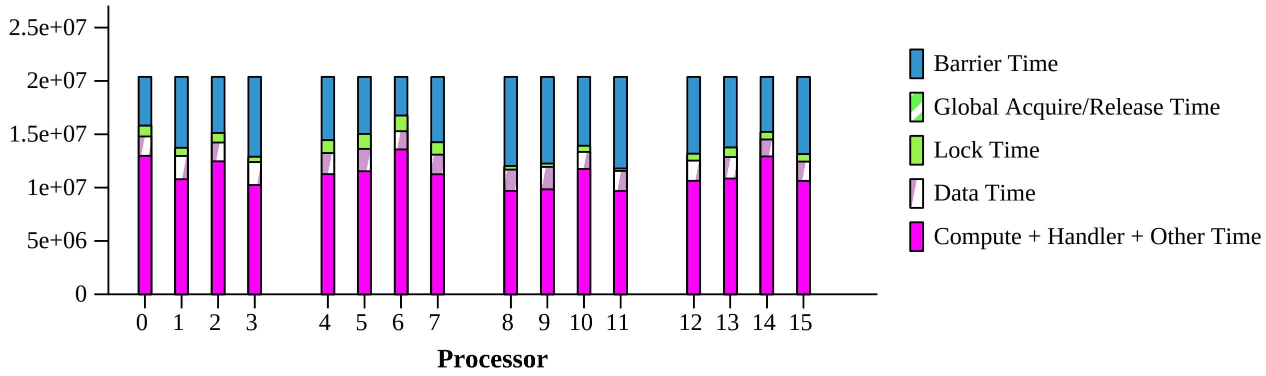

Figure 5.38: Execution time breakdown for Water-spatial, DW+RF+DD.

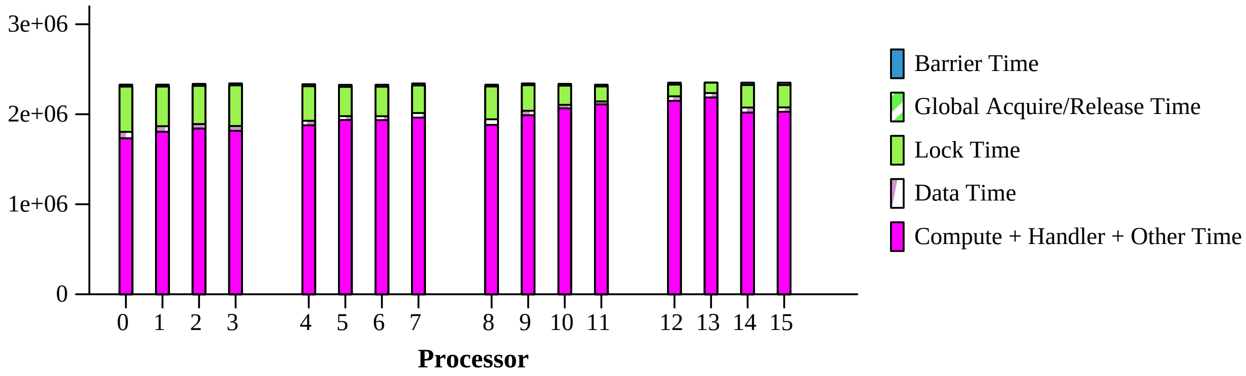

Figure 5.39: Execution time breakdown for Raytrace, Final.

Network interface locks (NIL): This protocol uses the locking support of the network interface. Since we explore the effect of the extensions cumulatively, this version includes all NI extensions and is the Final version of the protocol (SVM-NI). Compared to the previous version, the Final version substantially improves the performance of applications that use locks frequently, namely Raytrace, Volrend, Water-nsquared, and Water-spatial. Table 5.3 shows that lock time is reduced by (min 1.9%, avg 35.01%, max 62.7%).

In Water-nsquared the improvement is very significant due to the high frequency of locks; it mainly comes from the fact that no interrupts are necessary anymore, and

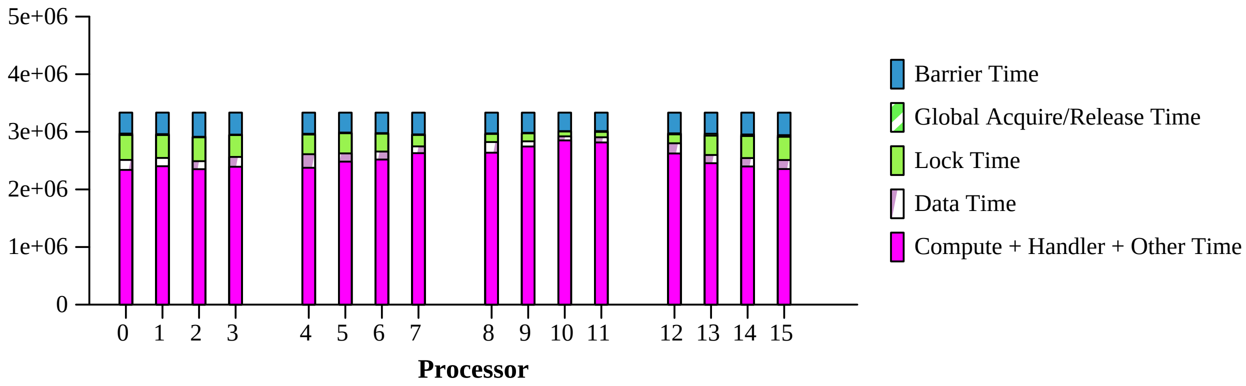

Figure 5.40: Execution time breakdown for Volrend, Final.

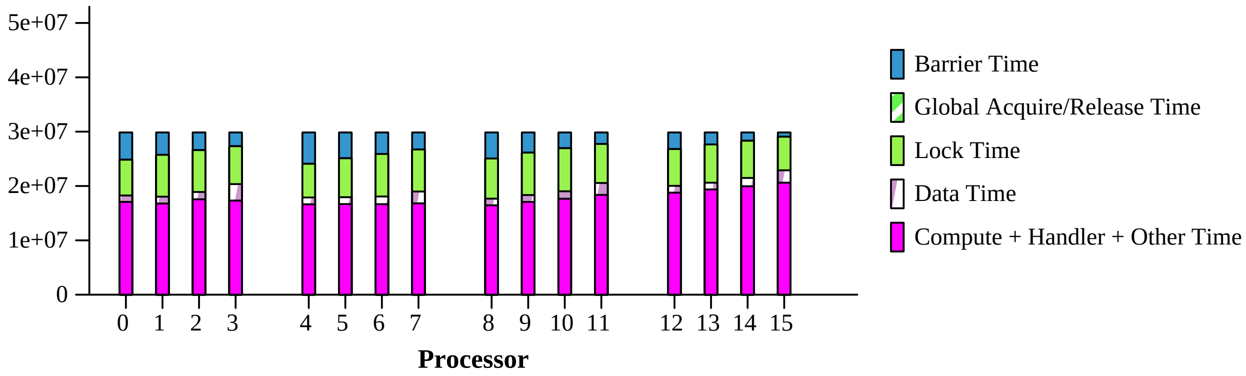

Figure 5.41: Execution time breakdown for Water-nsquared, Final.

also from the fact that lock messages do not need to be delivered to host memory. As discussed earlier, the latter results in shorter service times for lock messages since they need not wait for other messages to be delivered to the host first. The benefits in Water-spatial are smaller, since lock time is a less important component of the execution time.

In Raytrace, the more balanced compute time in the final version is due to the fact that this component includes not only the compute time, but the time spent in handling protocol requests as well. Using the lock mechanism in the network interface eliminates the need for lock protocol handlers. This results in more uniform execution,

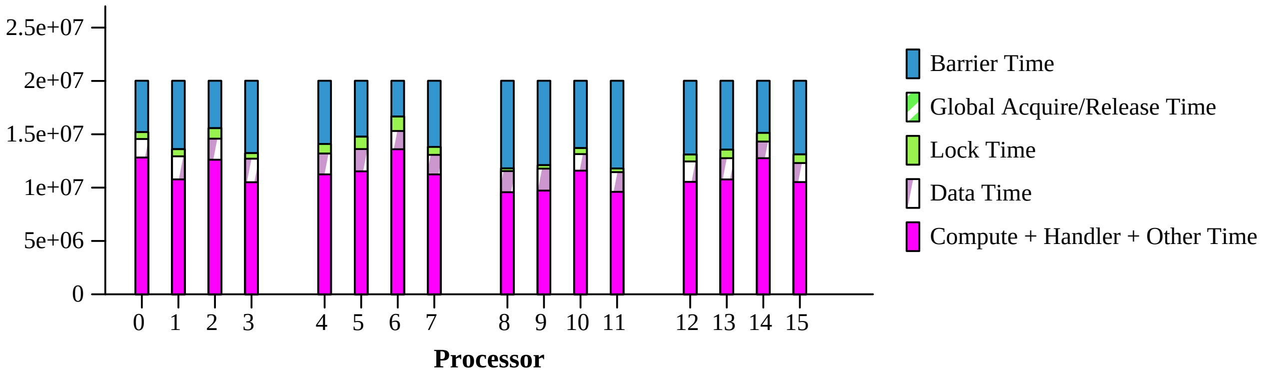

Figure 5.42: Execution time breakdown for Water-spatial, Final.

since only compute processes run on each processor.

Volrend is an interesting case, since with this Final protocol we manage to take advantage of task stealing and improve load balancing, as is demonstrated from the observed stealing pattern and the more balanced compute time. In previous studies [47], it was found that task stealing is not effective for SVM in Volrend because of the cost of locks. When task stealing was used in those cases, the cost of stealing a task from a process in a remote node was very high due to the high cost of locking. This made stealing hardly effective, since by the time the lock succeeded, there was little work left to be stolen for that frame. In the Final protocol however, the cost of locking is low and the processors that finish earlier can process stolen tasks before the owners of these tasks are done. This makes stealing effective, and leads to better load balancing, as seen in Figure 5.40.

Summary: We see that support for remote deposit and remote fetch operations in the NI has an effect on both the design and performance of SVM systems. By eliminating interrupts, they significantly enhance the ability of the network interface to support SVM effectively. They allow much more flexibility in the protocol design, since they provide to protocols a wider choice of solutions for accessing remote resources and allow them to better balance system costs, e.g., using either remote fetch or interrupts to fetch shared data. This often results in very substantial performance improvements. Table 5.3 shows that just by using the remote fetch operation the data wait time improves by (min 2.5%, avg 29.2%, max 45.3%)

Support for locking in the NI is also very useful when locking is frequent, often because it frees lock messages from having to travel all the way to the host processor and get stuck in queues. However, locking support requires more intrusive changes to the NI, and it may be more difficult to provide in network interfaces that do not have a programmable processor. Providing locking support in the NI improves application lock times by (min 1.9%, avg 35.01%, max 62.7%).

Although we have not yet evaluated this, we find that support for scatter–gather in the network interface can be beneficial for SVM protocols, for updating both protocol and shared data. However, since scatter–gather may require substantial processing on the network interface, depending on the semantics used, it may pose more stringent requirements on the network interface than the extensions we examine.

Overall, we see from Figure 5.2 that the Final version (SVM-NI) that leverages all our general–purpose NI extensions improves application performance by (min -20.16%, avg 37.68%, max 117.6%) across all applications. These are large improvements, and make a big difference to the end user, making SVM more competitive.

Figure 5.3 shows that the improvements in certain protocol costs do not always depend only on the extension that attacks the specific cost. For example, the introduction of direct diffs improves the performance of both lock and data wait times: lock time is reduced in Raytrace and data wait time is reduced in Barnes (rebuild). This happens because each change affects the behavior of the network interface, the pattern and the number of messages in the network, the queuing and contention, and (through asynchronous requests) the behavior of the protocol. Thus, it is not always straightforward to predict or quantify the improvement introduced by each extension, or determine the exact reasons for the results. To partially address this issue, in the next section we give a more general characterization of what happens in the network for the Base and the Final protocols, using the performance monitor.

5.6 Further Analysis

We have seen that using the NI in the manner we have done helps improve SVM performance substantially. However, Figure 5.43 shows that the resulting performance is still not quite where we would like it to be to compete with what efficient hardware coherence can achieve. In this section we use the performance monitoring tool in the communication layer [60] to answer two further questions. First, are there any performance problems caused by the changes we have made in the communication or protocol layers, which are simply overcome by the improvements, or have our changes only made improvements. We do this by examining the network interface activity in detail for the Base and Final protocol versions. Second, what are the main remaining sources of bottlenecks that we should go after to get the next leap in SVM performance.

5.6.1 Network interface activity and protocol tradeoffs

Two key potential differences in traffic in the Final protocol from the Base protocol are the use of eagerness and the related use of smaller protocol messages. As discussed earlier, the small messages achieve pipelining between host and NI, and the eagerness reduces acquire latency and helps eliminate interrupts in lock synchronization. However, traffic in the system is increased and this may result in increased contention. Since messages are asynchronous, the effects of contention may or may not be hidden

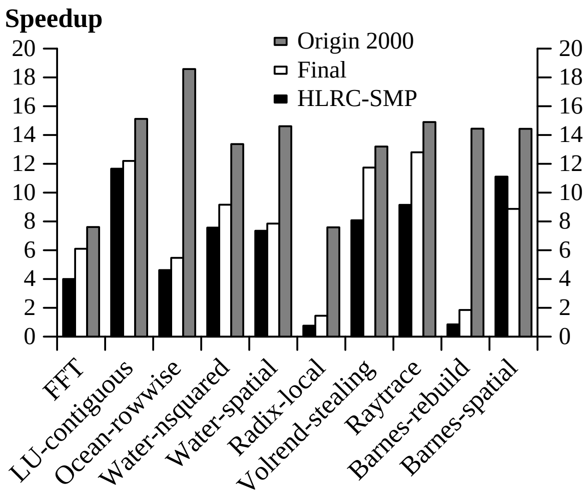

Figure 5.43: Application speedups (on 16 processors) for a hardware DSM machine (Origin–2000), compared with the Base and the Final protocols on the Myrinet system.

from the processors. Table 5.4 tries to quantify the amount of contention for certain applications in the Base and Final protocols, using the NI performance monitor. Each column represents one stage of the path from the sender to the receiver:

Post time is the time from the time the host processor posts a message, until the data for this message are in the NI. This stage may involve DMAing of data from host memory to the NI for big messages.

Send time is the time it takes the NI to send each packet to the network from the time it first appears in the NI. This stage includes queuing in the NI before a packet is sent to the network.

Transfer time is the time it takes each packet from when the first byte leaves the sending NI until the last byte is in the receiving NI.

Deliver time is the time it takes the receiving NI to deliver the packet to user memory from the moment it arrives in the network interface.

In all cases queuing and contention is included in the measurements. Note that in VMMC there is no explicit receive operation and thus there is no receive stage in the message transfer pipeline itself. Packets are delivered directly from the NI to the host memory. Even in the base protocol, destination processor involvement for message processing occurs after the message data have been placed somewhere in memory.

In Table 5.4 there are two numbers per stage for each application: one for the Base and one for the Final protocol. Each number is the ratio of the average time spent in the corresponding stage by small messages (up to 256 bytes) to the time that would be spent in the same stage in uncontended transfers. Thus, each number is a ratio that shows how much contention we have in that stage. We separate out small messages from large messages since we found that large messages behave very similarly in the two protocols and the contention for them in the NIs is very small.

Interestingly, we see that the Final version of the protocol greatly increases contention at the NI or network for almost all applications for all stages except the last one (data delivery to host memory). Moreover, for the same protocol version, applications that were found to perform poorly tend to have higher contention than others. The most apparent cases are Water-nsquared and Barnes-rebuild.

Although the Final version incurs higher contention in the network for small messages, the applications perform substantially better. This can be either because the improvements from other parts of the system, e.g., removing interrupts and scheduling problems from the processors, are higher, or that the Final system can tolerate contention because of its greater use of asynchronous communication. In the final protocol almost all operations are asynchronous and the host processor does not need to wait for the network operations to finish. The first reason is clearly a major factor, but we have not yet been able to determine the contribution of the latter. Nonetheless, the results suggest that we should indeed explore NI enhancements and protocol tradeoffs that reduce small messages, without introducing extra processing overheads to the host processors. NI enhancements include scatter–gather for invalidations and diffs, and protocol tradeoffs include using lazier forms of propagation as discussed earlier. Scatter–gather almost certainly should help, while the impact of protocol tradeoffs depends on added latency and the actual impact of the contention we observe on performance.

Application	Post	Send	Transfer	Deliver
Water-nsquared	1.7/10.4	2.2/13.8	1.9/13.2	3.2/5.1
Barnes-rebuild	-/8.6	-/12.6	-/12.4	-/6.0
Volrend	-/7.2	-/8.8	-/7.8	-/4.6
Raytrace	1.8/7.3	2.4/8.2	2.2/5.3	3.5/2.4
FFT	1.8/2.4	3.1/3.8	3.0/3.2	4.2/5.5
Ocean-continuous	1.5/-	2.4/-	2.0/-	4.3/-
Water-spatial	1.8/4.6	3.2/6.9	3.4/6.2	4.7/4.6
Radix	1.8/4.3	3.5/1.3	3.4/5.3	4.6/7.1
Barnes-spatial	1.9/3.2	3.6/5.5	3.6/4.3	5.2/5.8

Table 5.4: Ratios of average times over the uncontended time for each network or NI stage in the path from the sender to the receiver, for small messages in the Base and Final versions of the system.

5.6.2 Remaining performance bottlenecks for SVM

Finally, let us examine the key remaining bottlenecks in the Final protocol. The potential bottlenecks are memory bus contention and cache effects inside each SMP, lock time, barrier time and data wait time.

Figure 5.3 shows that for many applications the time spent in barrier operations is high. We divide barrier cost into two parts, the wait time at the barrier and the cost of protocol processing. This division is useful, since it shows whether major improvements require eliminating (or improving) protocol processing at barriers or better load balancing of computation, communication and protocol costs. Protocol costs at barriers are mainly due to changing the protection of pages, updating page

Application	Final	MBC	BT	BWT
FFT	6.1	7.6	8.46	13%
LU-contiguous	12.20	12.20	14.12	70%
Ocean-rowwise	5.47	7.5	12.85	50%
Water-nsquared	9.16	9.16	10.3	80%
Water-spatial	7.85	7.85	12.56	63%
Radix-local	1.45	1.65	4.52	6%
Volrend	11.74	12.00	13.26	65%
Raytrace	12.8	13.00	14.82	80%
Barnes-rebuild	1.85	2.03	2.69	81%
Barnes-spatial	8.87	8.87	14.43	18%

Table 5.5: Application speedups. The second column shows speedup for the Final version. The third and fourth columns are the speedups for each application if we exclude (a) memory bus contention (MBC), and (b) barrier cost. The last column (BWT) shows the percentage of barrier time that is spent waiting. The rest of the time is spent in protocol processing.

time–stamps and sending messages. These costs can be reduced if a lightweight thread model is used instead of the heavier–weight process model that we use for laziness [75]. The reason is that in the thread model, invalidations and page time–stamps need to be taken care of only once for all threads and this can be done in parallel. Of course, the thread model does reduce laziness of invalidations within an SMP, and there are other tradeoffs that must be explored [75]. Table 5.5 shows the effective speedups of the applications if we were to simply exclude the memory bus contention, the total barrier time and the barrier wait time, respectively, from the parallel execution profile. The second column in Table 5.5 shows the speedup for each application if we exclude the increase in local memory access costs compared to a system with one processor per node. Increasing the number of processors per node, may increase the contention on the memory bus, depending on how the bus and memory subsystem are designed. In the systems we use, running more than one processes per node, results in some applications in higher memory access times. For four applications, FFT, Ocean, Radix and Barnes-rebuild, the aggregate compute time in the parallel execution, in Final, increases when compared to the execution time of the sequential run. Actual compute time excluding local memory access shouldn't increase for these applications. For FFT, Ocean and Radix the increase is due to contention on the memory bus from four rather than one processor competing for the bus within a node (in runs with fewer processors per node this problem does not exist). This results in an increase in local compute time of 50%, 80% and 130% for each application respectively. This is specific to the design of the memory subsystem in the SMP nodes we use and may not be so much of an issue in other architectures.

The third column in Table 5.5 shows that, for most applications, excluding barrier costs results in very good speedups. The exceptions to this are FFT, Water-nsquared, Radix and Barnes-rebuild. The last column in Table 5.5 shows the percentage of barrier cost that is attributed to wait time (imbalances) as opposed to barrier processing cost. We see that for FFT, Radix, and Barnes-spatial most of the barrier cost is protocol processing time. More specifically, most of this cost is attributed to modifying the access right of pages with `mprotect` and to updating page time–stamps, so reducing these costs would be very valuable. Let us now look into the four applications for which barrier cost is not the key remaining bottleneck at this scale.

The remaining bottleneck in FFT is still the high data wait time, because of the low speed of the network interface and contention in the NI. If we compare the data wait time in FFT to what it would be with uncontended remote fetch operations, there is an increase of less than 30% in the cost. The rest of the cost is due to the uncontended cost of the remote fetch operation and can be addressed by faster networking hardware; protocol enhancements cannot do much about this. Even latency tolerance techniques might not work well due to the already existing contention and the fact that overall latency is dominated by end–points rather than transit time, which can be more easily overlapped.

Water-nsquared is limited by the high lock time as discussed earlier. For Radix the remaining bottleneck is the high data wait time due to the very high communication to computation ratio and false sharing of pages. The cost of `mprotect` is also very large here. Finally, in Barnes-rebuild the problem is synchronization and data wait time. These costs are both high and relatively imbalanced and they result in high barrier time as well.

Summary: The above analysis suggests that the two major areas of future effort to further improve SVM performance at this scale, other than the protocol tradeoffs and scatter–gather discussed earlier, are improving barrier processing (including and especially `mprotect` time, and further improving data wait time in some applications by enhancements (hardware or software) in the underlying communication layer.

5.7 Discussion and Conclusions

In this chapter we investigated how the capabilities of modern commodity network interfaces can be exploited to make the performance of SVM systems more competitive with hardware coherence. We proposed a set of extensions in the communication layer, that can significantly improve the performance of SVM by eliminating key bottlenecks in the system. This set of extensions impacts many different parts of the SVM protocols, with each extension targeting a significant overhead. However, there exist different network interface extensions that could target the same aspect of the protocol reducing some particular cost. We pick this set of extensions, since they are suited well to the type of SVM protocols and communication subsystem we use.

We demonstrated the benefits of our approach by implementing the set of extensions in the programmable network interface of the Myrinet network, and evaluating their impact on application performance on a network of Intel Pentium Pro SMPs. We chose to implement general–purpose data movement and synchronization mechanisms that are likely to be found (or can be implemented) in commodity network interfaces.

The software communication library that we use already provides us with a remote deposit interface, and we extend it by adding a remote fetch operation and by providing locking support. The resulting features in the communication layer (software and firmware) were used to alter the protocol layer accordingly. We presented an SVM protocol (SVM-NI) that uses these communication layer capabilities to completely avoid the need for asynchronous protocol processing by the host processors. In the final version of the protocol, whenever a processor wants to access some remote service, it does so through the general purpose extensions we add to the network interface. We also used a performance monitor, integrated in the communication layer, to understand the results and discover the sources of key remaining bottlenecks in the SVM system.

We find that although network contention increases in the final version of the system, due to the use of eager propagation and small messages, application performance improves by (min 4.6%, avg 44.1%, max 117.6%) across all applications. Similarly, the specific components of execution time targeted by the individual mechanisms improve substantially: data wait time improves by (min 2.5%, avg 29.2%, max 45.3%) and lock time by (min 1.9%, avg 35.01%, max 62.7%). The features we implemented in the NI also provide more flexibility in the choice of efficient protocol operations and management. Support for locking in the NI was also found to be very useful, though it requires more intrusive support. Different applications benefited greatly from different NI features, indicating that all three should be supported if possible.

Our results and detailed analysis have led us to several areas of future work. As mentioned above, while we used a set of protocol extensions that made sense given the communication enhancements and NI capabilities, other choices are possible too, introducing tradeoffs that need to be more fully explored. Support for other types of NI operations, e.g., remote atomic operations, can also be advantageous. The network interface and protocol extensions we examine show that our approach of using general purpose operations in the network interface to alleviate key system bottlenecks can indeed improve the performance of SVM significantly. However, there are more choices that need to be investigated, especially if the technology trends between the host system and the network interface change. Concrete examples of choices we did not explore and would be beneficial are a scatter–gather operation, and remote atomic operations. Finally, our results show that the next major features to attack to bring SVM closer in performance to efficient hardware–coherence, at least at this scale of 16 processors, are barrier processing time and especially the cost of `mprotect` operations to change page protections.

Chapter 6

Discussion and Conclusions

In this chapter we compare the simulation environment that we developed with the actual system implementation. We also present some future work and possible directions for continuing research in this and related areas. Finally we discuss our overall conclusions.

6.1 Simulator Validation

Our goal in this section is to present some evidence and intuition about the validity of the simulator that was used in Chapters 3 and 4. We compare the simulator with the system implementation that was presented in Chapter 5. The details of the simulation environment were presented in Section 3.5. The protocols that are used in the simulator and in the actual system were discussed in Chapters 3 and 5 respectively. In the next few sections we will discuss differences in the protocols used in the simulator and the real implementation, and then we will examine the comparative results in some detail.

Protocol layer discrepancies: From the protocol descriptions in Chapters 3 and 5 we see that the data structures used in the simulator and the implementation are somewhat different. The most notable difference is that the bins and the time–stamps are maintained on a per–processor basis in the simulator and on a per–node basis in the actual system. The simulator can maintain a finer notion of time, allowing for finer grain intervals, whereas the actual system reduces the size of control messages and the amount of protocol processing.

The policy for granting locks is different in the two systems. On the actual system, local locks are given priority over remote acquires. This policy may lead to starvation but it reduces certain costs and in practice works well. In the simulator, the back-end (Augmint) implements its own policy (FIFO) for granting locks to competing processes.

Certain operations cannot be modeled in detail in the simulator. Page invalidations for instance, are complicated operations that have varying costs, dependent on many factors (i.e. cache invalidations caused by the operating system, etc.). In the simulator, the cost for page invalidations is constant.

Finally, the protocol handlers are not simulated, since the simulator itself is not multi-threaded. The cost attributed to each handler is variable and it depends on the amount of work done. This however, is only an approximation of what is happening in the actual system.

Hardware layer discrepancies: Similarly, the hardware layer is not modeled in full detail. The processor is a one IPC processor with two levels of caches and a write buffer. The network interface is modeled as two queues for incoming and outgoing packets, and a state machines that incurs some overhead for each packet.

In the actual system both the components and their interactions are more complicated. Moreover the asynchronous nature of the system, as opposed to the synchronous nature of the simulator, introduces some discrepancy as well.

Basic costs: Table 6.1 shows some approximate numbers for certain basic operations in the two systems. Some of these numbers are difficult to measure with accuracy (especially on the actual system) and also they depend on the type of the experiment; the reader should use these as indicative of the cost for each operation.

We see that all costs are within a factor of two from each other, with the simulator being more optimistic. The only exception to this is the *bcopy* cost. The reason for this is that the actual system is limited by "real world" issues. The *bcopy* bandwidth is disproportionally low compared to the memory bus bandwidth. This is a limitation of the actual systems that we use, and the advantage of the simulator is that it does not have to deal with such issues.

Operation	Simulator	NI-SVM
Page transfer b/w	75 MBytes/s	50 MBytes/s
Page bcopy	300 MBytes/s	26 MBytes/s
Page fetch	75 μs	120 μs
Remote Lock Acquire	26 μs	30 μs
Barrier (16 procs)	350 μs	500 μs

Table 6.1: Basic costs for the simulator and the actual system (NI-SVM).

Application performance: We saw that the simulator differs in certain aspects from the actual system. Let us see now, given these differences, how well the simulator can match the results from the actual implementation. Figure 6.1 shows the speedups in the simulator and the implementation for some of the applications. We see that the speedups for most of the applications are relatively close, given the differences in the protocols and the underlying hardware layers. The only exceptions are Ocean-rowwise and Barnes-rebuild. The difference in Ocean-rowwise is due to two factors:

First, the fact that on the real system running four processes per node saturates the memory subsystem (for more information see Section 5.6). If fewer processes per node are used, this artifact does not occur and the speedup from the implementation is closer to the result from the simulator. Second, the fact that on the simulator, the sequential execution of Ocean-rowwise experiences a lot of capacity misses compared to the parallel execution, which results in the increase of the sequential execution time and consequently increase of the speedup. The difference in Barnes-rebuild is due to the more expensive locks in the simulator as will be explained next.

Lock time is in general higher in the simulator than the implementation. This is due to the order in which locks are acquired. In the implementation local acquires are given priority over remote requests. In the simulator the policy is, in some sense, more fair and it serves the request with a first come first serve (FIFO) policy regardless of whether the request is local or remote. This results in more remote acquires in the simulator.

Figures 6.2-6.15 present the execution time breakdowns for each of the applications. The execution time breakdowns are somewhat different in the two systems. The simulator gives detailed statistics about protocol execution time, so this is presented as a separate component. In the actual system it is difficult to obtain an accurate measurement for this cost, so it is included in the other costs.

For the most part, we see that the results are very close. In each case, the simulator captures the significant characteristics. In FFT (Figures 6.2, 6.3), LU (Figures 6.4, 6.5), Radix (Figures 6.10, 6.11), and Barnes-spatial (Figures 6.14, 6.15) the division of the time among compute time, data wait time, and barrier time is very similar in both systems. In Barnes-spatial, the simulator shows some imbalance in the computation time that is not there in the actual system, and this may be because of the way the initialization code is simulated. If this imbalance is removed (and the

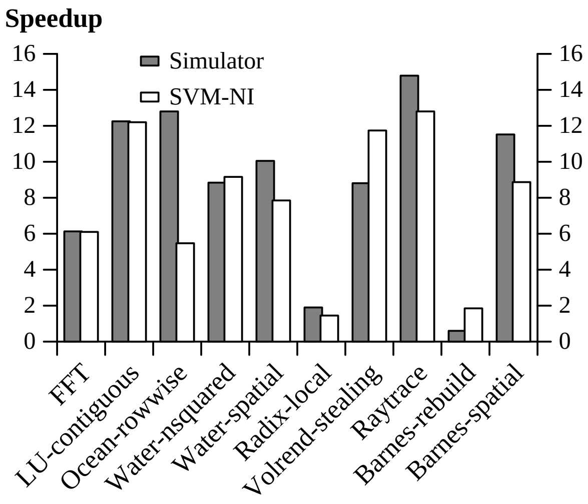

Figure 6.1: Speedup comparison for the SPLASH–2 applications between the simulator and the implementation. The protocols are somewhat different, and also some of the applications are different versions and some problem sizes differ.

corresponding imbalance in barrier time) the two cases match very well.

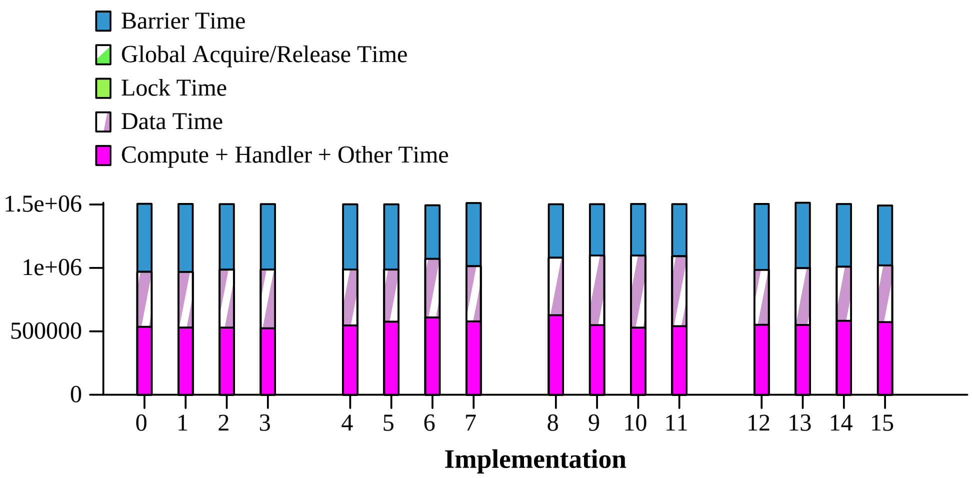

Figure 6.2: Implementation execution time breakdown for FFT.

In Water-nsquared (Figures 6.8, 6.9) the simulator matches very well with the implementation, showing that the bottleneck is the lock wait time. More significantly it captures the imbalances in lock wait time and the corresponding imbalances in barrier time. Similarly, in Ocean(Figures 6.6, 6.7) the simulator reveals that the

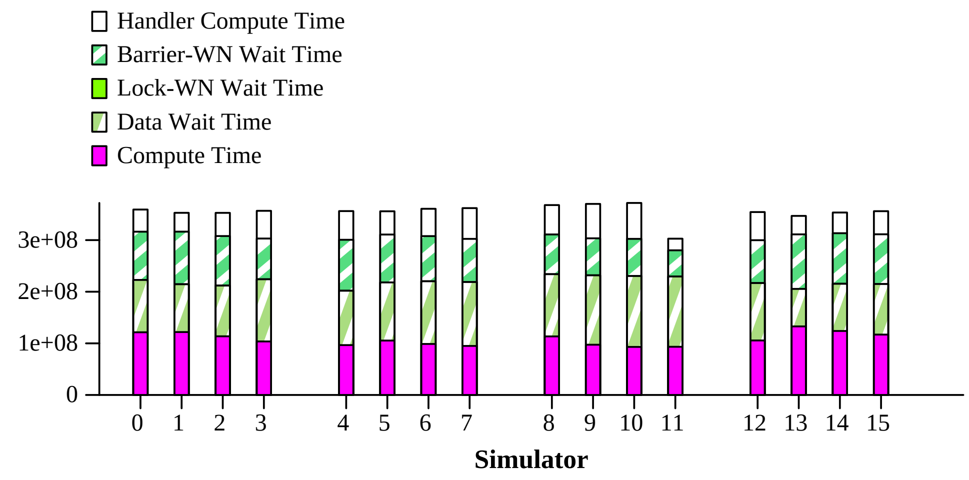

Figure 6.3: Simulator execution time breakdown for FFT.

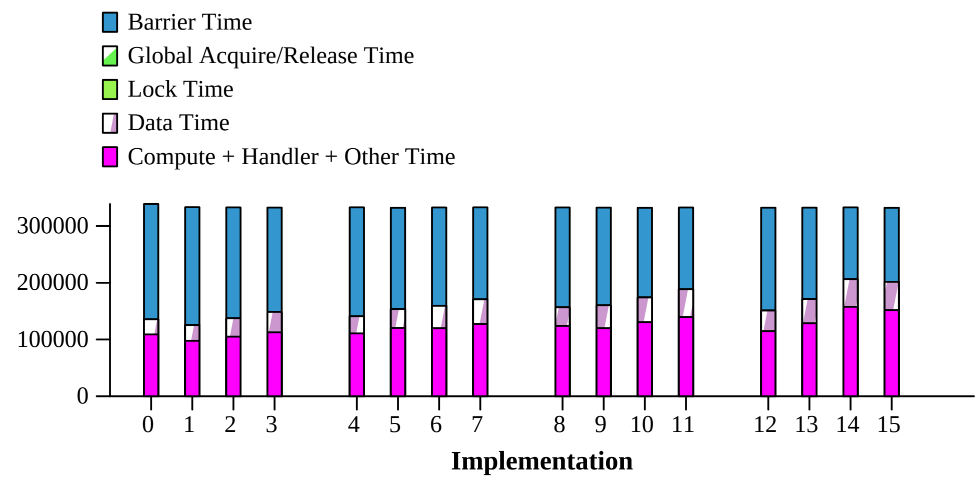

Figure 6.4: Implementation execution time breakdown for LU-contiguous.

bottleneck is the barrier cost, although to a smaller extent than in the actual system. In Barnes-rebuild (Figures 6.12, 6.13) the simulator reveals that the bottleneck is lock synchronization, however, at a bigger extent than in the actual system.

We see that the simulator is successful at modeling the system behavior. In certain cases, however, the results are more optimistic, which is in fact what led us to investigate and try to resolve the bottlenecks in the actual system. Overall, we found detailed architectural simulation to be an invaluable tool in understanding system behavior and application–system interactions that are difficult to obtain from

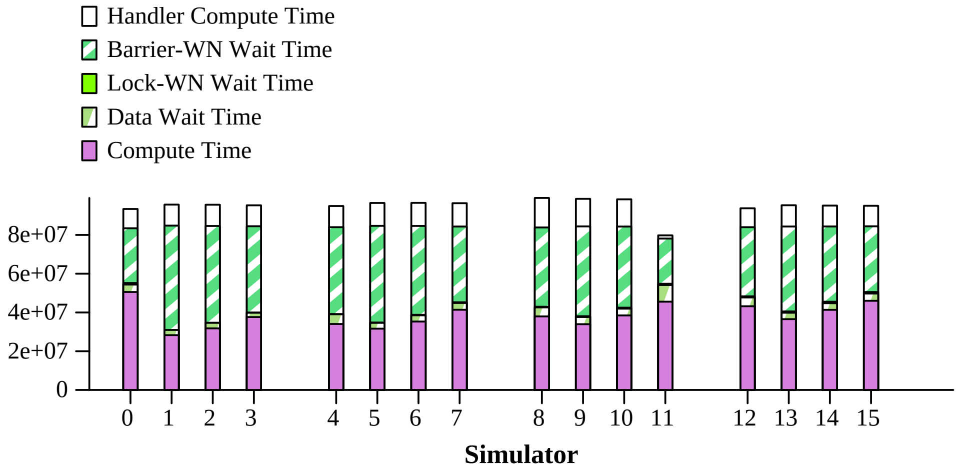

Figure 6.5: Simulation execution time breakdown for LU-contiguous.

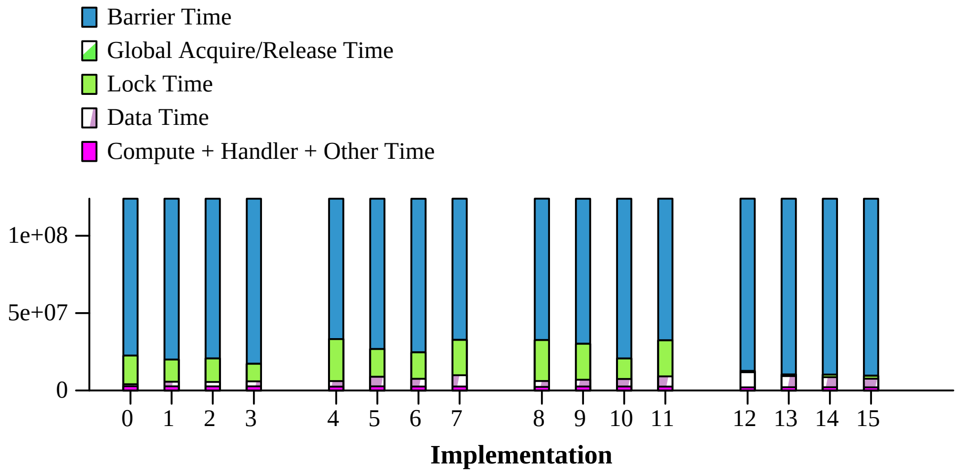

Figure 6.6: Implementation execution time breakdown for Ocean-contiguous.

the real implementation, and in comparing with the real implementation results to diagnose the causes of conflicts in the latter.

6.2 Future Work

The areas that this dissertation overlaps with are network interfaces and network interconnects for parallel systems, shared virtual memory as a programming abstraction, and architectural simulation, modeling and analysis. Outstanding issues in these

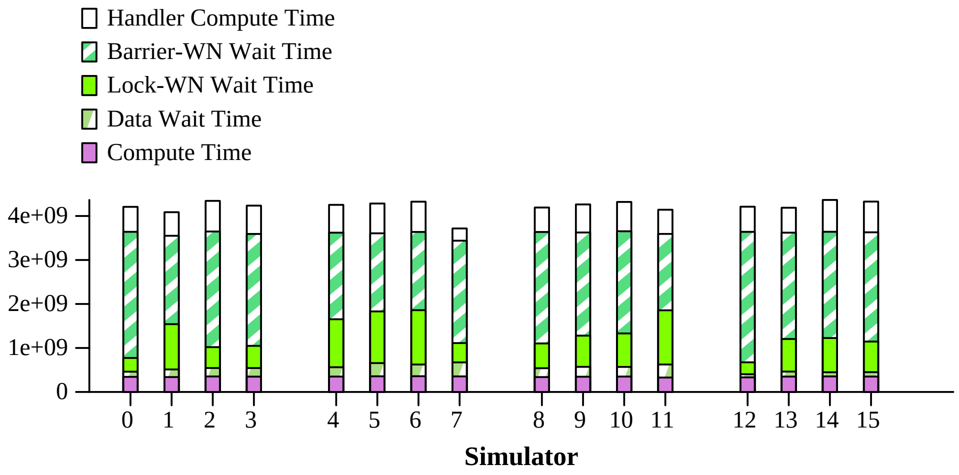

Figure 6.7: Simulation execution time breakdown for Ocean-contiguous.

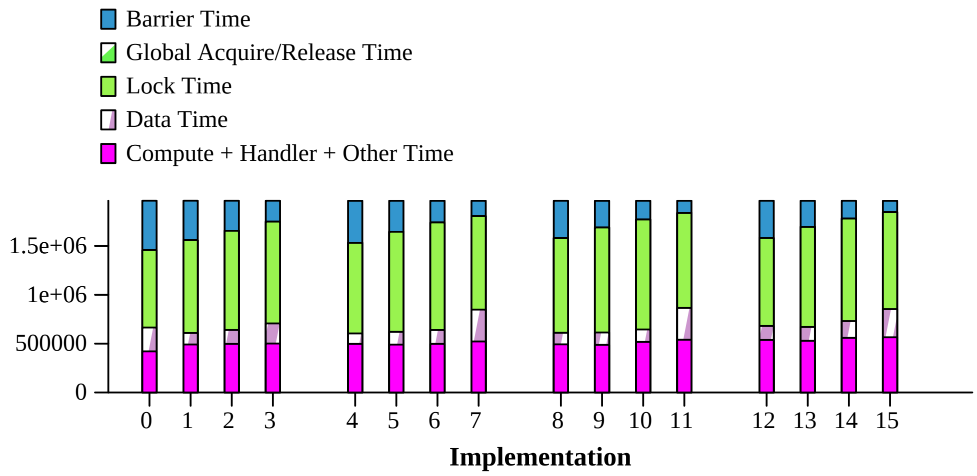

Figure 6.8: Implementation execution time breakdown for Water-nsquared.

areas include:

In the area of network interfaces and network interconnects, the work done focuses in providing a high level of performance and functionality with communication hardware (system area networks) that is between the very special supercomputer networks and the more general purpose local area networks. However, the system area networks that we use, connect systems at the I/O bus level, they employ a certain processor–NI interface, and provide some level of functionality. More work is needed to figure out whether the network interfaces for this type of networks can/should be

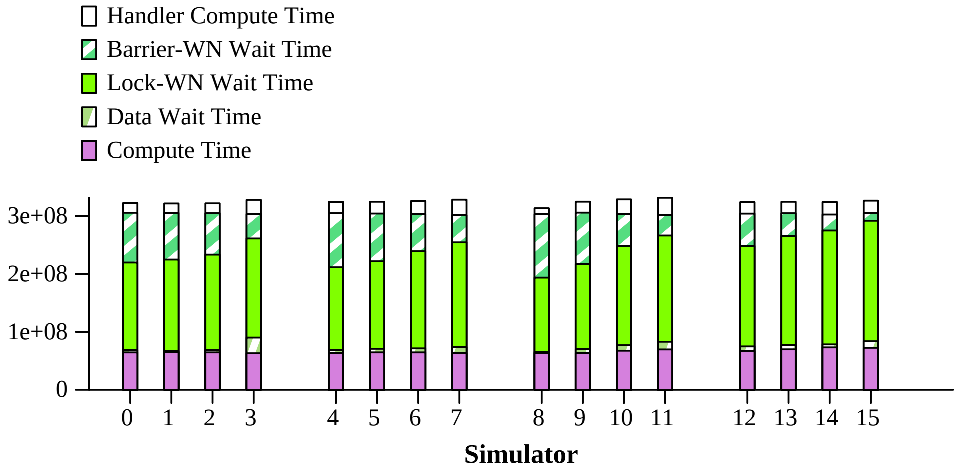

Figure 6.9: Simulation execution time breakdown for Water-nsquared.

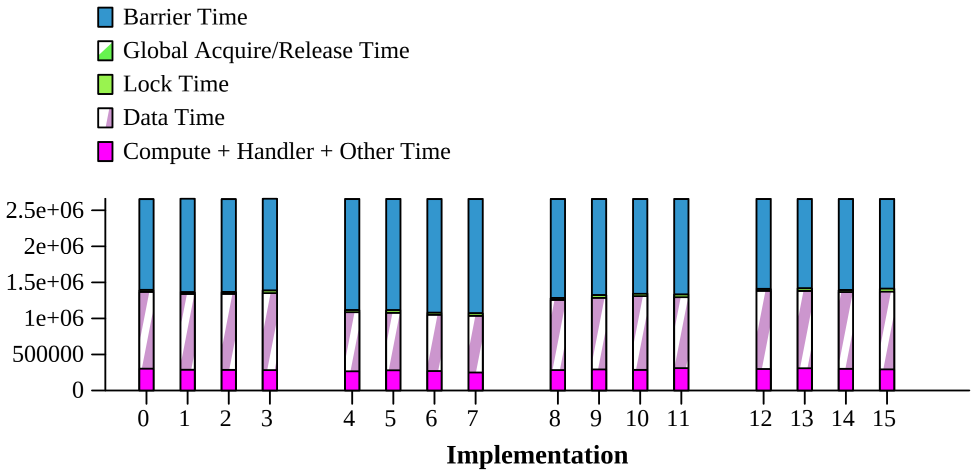

Figure 6.10: Implementation execution time breakdown for Radix-original.

closer to the processor, i.e., on the memory bus, whether the functionality should be fixed, or the network interfaces should be programmable because of the diverse needs they have to serve, and what the interface to the processor should be. The benefits of the proposed approaches are significant for system area networks, it is not clear if the same approaches apply to other existing or new types of networks that may span the design spaces between multicomputer and system area networks and system area networks and local area networks. Moreover, it is not clear yet what the behavior of these system area networks is, when the number of nodes increases to the order of

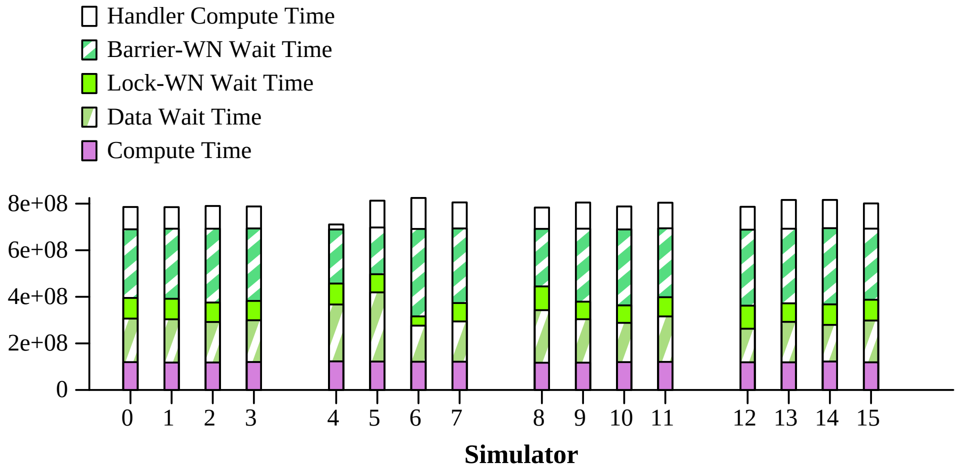

Figure 6.11: Simulation execution time breakdown for Radix-original.

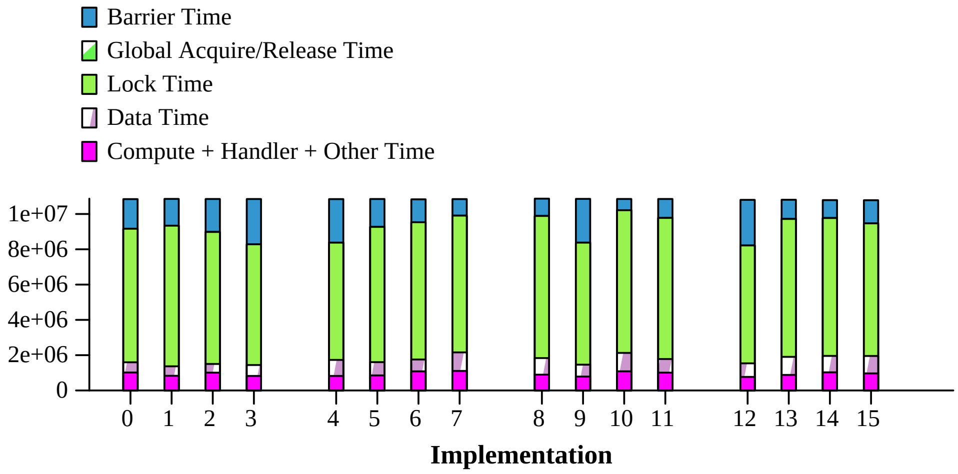

Figure 6.12: Implementation execution time breakdown for Barnes-rebuild.

hundreds of nodes and they cover bigger geographical areas. In short, it is not clear what the relation of these networks is with the more traditional supercomputer and local area networks, in terms of performance, functionality, and scalability.

In the area of shared virtual memory, this dissertation improved the performance of SVM on clusters of workstations connected with a system area network. Many applications that performed in mediocre ways, exhibit now either good or "reasonable" speedups. Given this improvements in system performance, the questions that become more important are: i) how does the system scale with more processors per

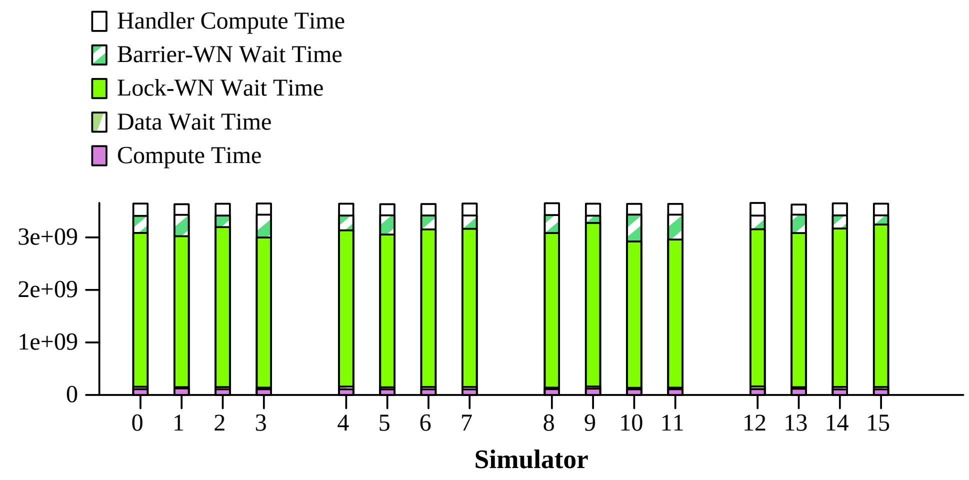

Figure 6.13: Simulation execution time breakdown for Barnes-rebuild.

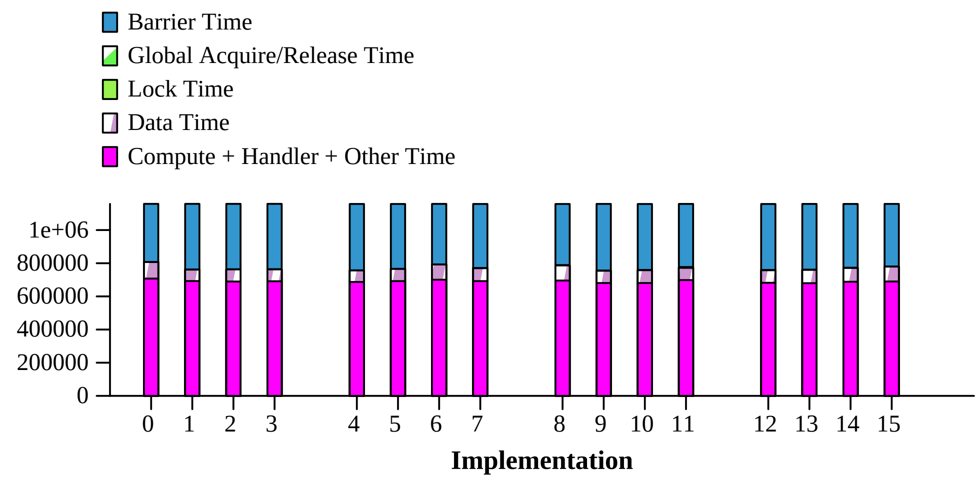

Figure 6.14: Implementation execution time breakdown for Barnes-spatial.

node, more nodes (SMPs or DSMs), and bigger problem sizes, and ii) how does the system behave with more commercially-oriented applications.

In the area of architectural simulation, the most pressing question seems to be how to simulate the parallel architectures of the future. So far, most of the effort in simulation has focused in (i) examining alternatives for current systems or systems that were about to be built, and (ii) providing a means for other research to continue while a system was being built. Simulating systems that will be built a few years in the future with very big caches and memories that run complicated communication

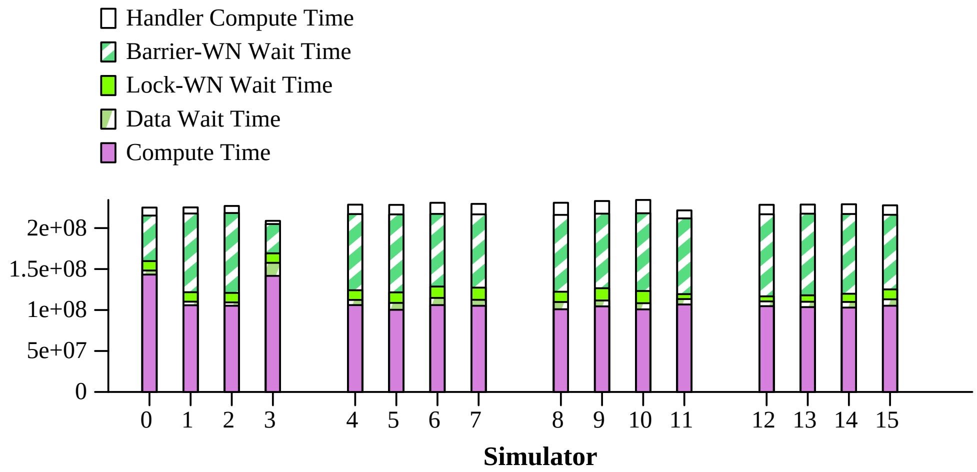

Figure 6.15: Simulation execution time breakdown for Barnes-spatial.

protocols and application problem sizes that are beyond our reach today, would give us valuable insights on the future trends in parallel computer architecture. However, it is, if not impossible, at least practically impossible to simulate these architectures on today's systems because of the required state that needs to be maintained and manipulated. Methods that overcome these problems would be invaluable in advancing the state of the art in architectural simulation.

At a higher level, the issues of operating systems for clusters interconnected with system area networks, as well as all the issues related with reliability, robustness and fault tolerance of this architecture are still open. Most of the work in the area tried to by–pass the operating system, since it is usually the source of difficult–to–control overheads. However, there are system aspects that are difficult to deal with if the operating system is ignored. Management, maintenance, transparency, extensibility are some of these. Similarly, certain applications, especially some types of commercial applications, require systems that not only provide good performance, but fulfill strict requirements on robustness and reliability as well.

In general, more work is needed before we can fully understand if clusters of workstations interconnected with system area networks are a viable direction to parallel computing. At the same time, more work is required to improve the system analysis and evaluation methodology.

6.3 Conclusions

In this dissertation we first present an efficient, reliable and flexible communication layer for system area networks. We demonstrate the validity of our approach by describing the design and implementation of the virtual memory–mapped communication model (VMMC) [15, 25] on a Myrinet network of PCI–based PCs. VMMC is a communication model that was proposed for the SHRIMP multicomputer [16] and provides direct data transfer between the sender's and receiver's virtual address spaces. It eliminates operating system involvement in communication, provides full protection, supports user–level buffer management and zero–copy protocols, and minimizes software communication overhead. Moreover we propose and implement reliable communication in the firmware that runs in the network interface at very low cost. Finally, we propose a new scheme for dynamically determining the network topology. This mechanism relies on the retransmission mechanism that is used to provide reliable communication. The implementation of VMMC on Myrinet achieves about 11 μs one–way latency and provides about 100 MBytes/s user–to–user bandwidth. This demonstrates that low–latency, high–bandwidth communication can be provided with off–the–shelf hardware components. The cost of reliable communication is minimal; less than 2.5μs for latency and less than 10% for all different types of bandwidth. Similarly, since the support for dynamic system configuration is not on the common path, there is no effect on system performance when no changes in the system topology occur.

Second we address the issues of using SMPs as nodes in the cluster and the implications to protocol design and performance for all–software protocols as well as protocols that use hardware support for automatic write propagation with or without write–back caches. To quantify the benefits of each approach, we start from a recent and particularly promising form of SVM protocols is the class of so–called *home–based* protocols (HLRC, AURC) [44, 45, 46, 100]. We find that: (i) the performance of both AURC and HLRC improves substantially with the use of SMP rather than uniprocessor nodes in five of the ten applications. In three applications there is a smaller improvement (or they perform the same as in the uniprocessor node case) and for the other two results differ across all–software and automatic update protocols, with the latter performing worse with SMPs than with uniprocessors; (ii) AU support with write–through caches does significantly improve performance of some irregular applications, especially when diff–related costs dilate critical sections and increase serialization, but it can also hurt substantially in some cases; (iii) AU support with write–back caches solves the problems caused by the increased traffic and performs best, but is very intrusive into the underlying node; (iv) overall, the benefits are quite limited with commodity NIs with their higher occupancies per packet, and NIs customized for AU packet generation may be important for using AU effectively.

Third we investigate the importance of communication architecture parameters on application performance and reveal the most important system bottlenecks with respect to the communication architecture. We find that the most important system cost to improve is the overhead of generating and delivering interrupts. Improving network interface (and I/O bus) bandwidth relative to processor speed helps some bandwidth–bound applications, but currently available ratios of bandwidth to processor speed are already adequate for many others. Surprisingly, neither the processor overhead for handling messages nor the occupancy of the communication interface in preparing and pushing packets through the network appear to require much improvement.

Finally we propose and evaluate extensions in protocol, the software communication layer, and the underlying network interface that substantially enhance the performance of shared virtual memory on clusters of SMPs. We examine how the protocol layer can take advantage of various mechanisms in the communication layer and be restructured accordingly. The final protocol, SVM-NI, eliminates the need for interrupts and asynchronous protocol handling. We demonstrate that substantial improvements in performance can indeed be achieved, and find that different applications need different mechanisms among the ones we use. Overall application performance improves significantly and individual components of execution time targeted by each mechanism are reduced by even higher percentages.

At a higher level, this dissertation deals with issues related to making clusters of workstations a viable architecture for parallel computing. It deals with issues at the communication and the programming abstraction layer. At the communication layer it advances the state of the art by demonstrating that rich functionality and high performance can co–exist on top of commodity hardware. At the protocol layer it investigates the system bottlenecks and proposes new protocols that alleviate these bottlenecks by taking advantage of the node and the network architectures. These results are very encouraging, since they contribute to reducing the gap between the performance of this and previous architectures for parallel computing, however, at a much lower cost. The main advantages of this, cluster architecture is its ability to track the technology curves well and the promise for scalability and robustness. These make clusters a very appealing alternative to traditional supercomputers and mainframes for both scientific and commercial use. However, these two directions, scalability and robustness, that are essential either for large scale, scientific–like parallel computation, or for fault tolerant commercial systems are still open. Thus besides trying to further improve system performance, as mentioned in the future work section, more work is required to understand the behavior of clusters with respect to scalability and robustness and to solve the existing problems. Investigating these issues, is vital for the success of clusters as a parallel architecture and they are now, after making significant progress with the performance and functionality issues, the next pressing problems that need to be addressed.

Bibliography

[1] E. Arnould, F. Bitz, E. Cooper, H. Kung, R. Sansom, and P. Steenkiste. The design of nectar: A network backplane for heterogeneous multicomptuers. In *The 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, Apr. 1989.

[2] D. H. Bailey. FFTs in External or Hierarchical Memories. *Journal of Supercomputing*, 4:23–25, 1990.

[3] J. E. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, December 1995.

[5] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Seattle, Washington, Mar. 1990.

[6] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE COMPCON '93 Conference*, Feb. 1993.

[7] R. Bianchini, L. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software dsms. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[8] A. Bilas, L. Iftode, D. Martin, and J. P. Singh. Shared virtual memory across SMP nodes using automatic update: Protocols and performance. Technical Report TR-517-96, Princeton, NJ, Mar. 1996.

[9] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. In *IMA Workshop on Parallel Algorithms and Parallel Systems*, Nov. 1996.

[10] A. Bilas, L. Iftode, and J. P. Singh. Evaluation of hardware support for shared virtual memory clusters. In *The 12th ACM International Conference on Supercomputing (ICS'98)*, July 1998.

[11] A. Bilas, C. Liao, and J. P. Singh. Network interface support for shared virtual memory on clusters. Technical Report TR-579-98, Computer Science Department, Princeton University, Princeton, NJ-08544, Mar. 1998.

[12] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *In Proceedings of Supercomputing 97, San Jose, CA*, November 1997.

[13] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[14] M. Blumrich, C. Dubnick, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[15] M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual memory mapped network interfaces. *IEEE Micro*, 15(1):21–28, Feb. 1995.

[16] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[17] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[18] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[19] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, Apr. 1994.

[20] D. Culler, L. Liu, R. P. Martin, and C. Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, 1996.

[21] D. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, 1998.

[22] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[23] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[24] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pages 388–396, April 1997.

[25] C. Dubnicki, L. Iftode, E. Felten, and K. Li. Software support for virtual memory-mapped communication. In *Proceedings of the 1996 International Parallel Processing Symposium*, Apr. 1996.

[26] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, Aug. 1997.

[27] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[28] A. Erlichson, B. Nayfeh, J. P. Singh, and K. Olukotun. The benefits of clustering in shared address space multiprocessors: An applications-driven investigation. In *Supercomputing '95*, pages 176–186, 1995.

[29] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1996.

[30] G. E. B. et al. A comparison of sorting algorithms for the connection machine cm-2. In *Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[31] B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *The 3nd IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, 1997.

[32] E. Felten, R. Alpert, A. Bilas, M. Blumrich, D. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[33] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–259, Apr. 1991.

[34] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory letency using dynamic scheduling in shared-memory multiprocessors. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Poster, May 1992.

[35] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[36] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[37] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb. 1996.

[38] C. J. Glass and L. M. Ni. The turn model for adaptive routing. *Journal of the ACM*, 41(5):874–902, Sept. 1994.

[39] N. Hardavellas, G. C. Hunt, S. Ioannidis, R. Stets, S. Dwarkadas, L. Kontothanassis, and M. L. Scott. Efficient use of memory-mapped network interfaces for shared memory computing. *Newsletter of the IEEE CS Technical Committee on Computer Architecture*, pages 28–33, Mar. 1997.

[40] L. Hernquist. Hierarchical N-body methods. *Computer Physics Communications*, 48:107–115, 1988.

[41] C. Holt, M. Heinrich, J. P. Singh, , and J. L. Hennessy. The effects of latency and occupancy on the performance of dsm multiprocessors. Technical Report CSL-TR-95-xxx, Stanford University, 1995.

[42] C. Holt, J. P. Singh, and J. Hennessy. Architectural and application bottlenecks in scalable DSM multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[43] R. W. Horst and D. Garcia. ServerNet SAN I/O Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, Aug. 1997.

[44] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[45] L. Iftode, J. P. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[46] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[47] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[48] D. Jiang and J. P. Singh. A methodology and an evaluation of the sgi origin2000. In *In Proceedings of ACM Sigmetrics / Performance'98, Madison, Wisconsin*, June 1998.

[49] M. Karlsson and P. Stenstrom. Performance evaluation of cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers. In

The 2nd IEEE Symposium on High-Performance Computer Architecture, Feb. 1996.

[50] M. Katevenis. Telegraphos: High-speed communication architecture for parallel and distribut ed computer systems. Technical Report No. 123, ICS-FORTH, Heraklio, Crete, Greece, May 1994.

[51] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, Jan. 1994.

[52] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[53] L. I. Kontothanasis, M. L. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing '95*, Nov. 1995.

[54] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 157–169, June 1997.

[55] L. I. Kontothanassis and M. L. Scott. High performance software coherence for current and future architectures. *Journal of Parallel and Distributed Computing*, Nov. 1995.

[56] L. I. Kontothanassis and M. L. Scott. Using memory-mapped network interfaces to improve t he performance of distributed shared memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[57] J. P. Laudon and D. Lenoski. The sgi origin2000: A scalable cc-numa server. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, June 1997.

[58] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Proceedings of COMPCON'90*, pages 62–67, 1990.

[59] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[60] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a myrinet-connected shrimp cluster. Submitted for publication, 1998.

[61] S. Lumetta, A. Mainwaring, and D. Culler. Multi-protocol active messages on a cluster of SMP's. In *In Proceedings of Supercomputing 97, San Jose, CA*, November 1997.

[62] A. M. Mainwaring, B. N. Chun, S. Schleimer, and D. S. Wilkerson. System area network mapping. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 116–126, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.

[63] A. M. Mainwaring and D. E. Culler. Active Message applications interface and communication subsystem organization. Technical Report CSD-96-918, Computer Science Division, University of California at Berkeley, 1995.

[64] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effect of communication latency, overhead, and bandwidth on a cluster architecture. Technical Report CSD-96-925, Berkeley, Nov. 1996.

[65] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, Oct. 1992.

[66] A. G. Nowatzyk, M. C. Browne, E. J. Kelly, and M. Parkin. S-Connect: from networks of workstations to supercomputer peformance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 71–82, New York, June 22–24 1995. ACM Press.

[67] S. S. Owicki and A. R. Karlin. Factors in the performance of the AN1 computer network. Technical Report 88, DEC System Research Center, 130 Lytton Ave., Palo Alto, CA 94301, June 1992.

[68] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The Fast Messages (FM) 2.0 streaming interface. Usenix'97, 1996.

[69] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for myrinet. In *Supercomputing '95*, 1995.

[70] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.0*, Apr. 1993.

[71] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In *In PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA*, March 30 – April 3 1998.

[72] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, Apr. 1994.

[73] T. L. Rodeheffer and M. D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 183–97. Association for Computing Machinery SIGOPS, Oct. 1991.

[74] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *USENIX '97*, 1997.

[75] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design, simulations, implementation and performance. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture, Las Vegas*, February 1998.

[76] D. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Oct. 1997.

[77] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[78] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 125–136, Jan. 1998.

[79] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access for distributed shared memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, Oct. 1994.

[80] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report 59, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, Apr. 1990.

[81] A. Sharma, A. T. Nguyen, J. Torellas, M. Michael, and J. Carbajal. Augmint: a multiprocessor simulation environment for Intel x86 architectures. Technical report, University of Illinois at Urbana-Champaign, March 1996.

[82] J. P. Singh, A. Gupta, and J. L. Hennessy. Implications of hierarchical N-body techniques for multiprocessor architecture. *ACM Transactions on Computer Systems*, May 1995. To appear. Early version available as Stanford Univeristy Tech. Report no. CSL-TR-92-506, January 1992.

[83] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(6), june 1994.

[84] J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[85] J. P. Singh and J. L. Hennessy. Parallelism, locality and scaling in a molecular dynamics simulation. To appear as Stanford University Technical Report, 1992.

[86] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical N-body techniques for multiprocessor architecture. Submitted for publication, 1991.

[87] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, July 1993.

[88] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 26(7):42–50, july 1993.

[89] J. P. Singh, W. D. Weber, and A. Gupta. SPLASH: stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[90] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *The 3nd IEEE Symposium on High-Performance Computer Architecture*, Feb 1997.

[91] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[92] D. Stodolsky, J. B. Chen, and B. Bershad. Fast interrupt priority management in operating system kernels. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures: September 20–21, 1993, San Diego, California, USA*, pages 105–110, Berkeley, CA, USA, Sept. 1993. USENIX.

[93] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.

[94] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst.*, 11(2):179–203, May 1993.

[95] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient support for multicomputing on atm networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, Apr. 1993.

[96] R. Thekkath, A. P. Singh, J. P. Singh, J. Hennessy, and S. John. An application-driven evaluation of the convex exemplar spp-1200. In *Proceedings of the International Parallel Processing Symposium*, April 1997.

[97] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

[98] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating message-passing in cache-coherent multiprocessors. In *Proceedings of Architectural Support For Programming Languages and Operating Systems*, 1994.

[99] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: a multigrain shared memory system. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[100] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, Oct. 1996.

[101] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. Toonen, I. Schoinas, M. Hill, and D. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.