In *Proceedings of the 1995 International Conference on Parallel Processing*, pages I–194-201, August 1995.

The Quest for a Zero Overhead Shared Memory Parallel Machine*

Gautam Shah Aman Singla Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280.
{gautam, aman, rama}@cc.gatech.edu

Abstract – *In this paper we present a new approach to benchmark the performance of shared memory systems. This approach focuses on recognizing how far off the performance of a given memory system is from a realistic ideal parallel machine. We define such a realistic machine model, called the* z-machine*, that accounts for the inherent communication costs in an application by tracking the data flow in the application. The z-machine is incorporated into an execution-driven simulation framework and is used as a reference for benchmarking different memory systems. The components of the overheads in these memory systems are identified and quantified for four applications. Using the z-machine performance as the standard to strive for we discuss the implications of the performance results and suggest architectural trends to pursue for realizing a zero overhead shared memory machine.*

1 Introduction

Realization of scalable shared memory machines is the quest of several ongoing research projects both in industry and academia. A collection of nodes interconnected via some kind of network, with each node having a piece of the globally shared memory is the common hardware model assumed. The cost of communication experienced on any realization of such a model is an important factor limiting the performance and scalability of a system[1]. Several techniques including relaxed memory consistency models, coherent caches, explicit communication primitives, and multithreading have been proposed to reduce and/or tolerate these communication overheads that arise due to shared memory accesses that traverse the network. In general the goal of all such techniques is to make the parallel machine appear as a zero overhead machine from the point of view of an application. It is usually recognized that no one technique is universally applicable for reducing or tolerating the communication overheads in all situations [8].

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

[1]We use the term system to denote an algorithm-architecture pair.

There have been several recent studies in separating the overheads seen in the execution of an application on a parallel architecture [4, 13]. These studies shed important light on categorizing the sources of overhead, and the relative advantage of a particular technique in reducing a particular overhead category. For example, Sivasubramaniam et al. [13] break-down the overheads into algorithmic (i.e. inherent in the application such as serial component), and interaction (i.e. due to the communication and system overheads seen by the application when mapped onto a given architecture). All the techniques for latency reduction and tolerance attempt to shave off this interaction overhead. However, we would like to be able to quantify the amount of communication delay that is *inevitable* in the execution of the application on any given architecture. Any communication that is seen over and above this "realistic ideal" is an overhead, and we refer to it as *communication overhead*. Armed with this knowledge, we can then benchmark the overheads introduced by any given memory system which is meant to aid the communication in an application. Clearly, if an application has dynamic communication pattern we have to resort to execution-driven simulation to determine the communication that is inevitable in the application.

The quest for a machine model that has zero communication overhead from the point of view of an application is the goal of this work. PRAM [16] has been used quite successfully as a vehicle for parallel algorithm design. In [13], it is shown how PRAM could be used as a vehicle for determining the algorithmic overhead in an application as well by using the PRAM in an execution-driven framework. Unfortunately, the PRAM model assigns unit cost for all memory accesses and does not take into account even the realistic communication costs. Thus it is difficult to realize hardware models which behave like a PRAM and to have faith in PRAM as a performance evaluation tool.

We develop a *base machine model* that achieves this objective of quantifying the inherent communication in an application (Section 2). Then, we present an implementation of this base machine model in the SPASM simulation framework (Section 3). Four different shared memory

systems (Section 4) are evaluated to identify the overheads they introduce over the communication required in the base machine (Section 5).

The primary contribution of this work is the base machine model that allows benchmarking the overheads introduced by different memory systems. Another contribution is the breakdown of overheads seen in different memory systems in the context of four different applications. The framework also leads to a better understanding of the application demands, and the merits and demerits of various features of the different memory systems. Such an understanding can help in deriving architectural mechanisms in shared memory systems that strive to perform like the base machine.

2 Base Machine Model

2.1 Communication in an Application

Before we identify the overheads that may be introduced by a shared memory system, we must understand the communication requirements of an application. Given a specific mapping of an application onto an architecture, its communication requirements are clearly defined. This is precisely the data movements warranted by the producer-consumer relationships between the parallel threads of the application. In order to keep track of this data flow relationship various memory systems tag on additional communication costs. There are a couple of ways we can tackle the issue of reducing communication costs. The first is by reducing the overheads introduced by the memory systems. The other is by providing ways to overlap computation with communication. The goodness of a memory system depends on its ability to keep the additional costs low and to maximize the computation-communication overlap. We therefore need a model that can give a measure of this goodness.

Given the producer-consumer relationships, applications use synchronization operations as firewalls to ensure that the program is data-race free. The inherent cost to achieve process coordination, depends on the support available for synchronization. Any additional cost that the memory system associates with the synchronization points is baggage added by the memory system to achieve its goals and thus should be looked upon as overhead.

Some of the communication requirements of an application can be determined statically based on the control flow, while the others can only be determined dynamically. Depending on the latency for remote communication on the interconnection network, the requirements will translate to a certain *lower bound* on the communication cost inherent in the application. Our goal in striving for a base machine model, is to be able to quantify this communication cost.

Figure 1 illustrates what we mean by inherent communication cost and overheads. Consider the case in which processor $P1$ writes a value at time $t1$. The reads corresponding to this write are issued by processors $P0$ and $P2$ at times $t2$ and $t6$ respectively. If we take into account the communication latencies only, the value written by $P1$ will propagate to processors $P0$ and $P2$ at times $t5$ and $t3$ respectively. Thus processor $P2$ experiences an inherent communication cost of ($t3$ - $t2$), whereas there is no inherent communication cost associated with the read by $P0$ at time $t6$. Note that in the case of $P0$, there is inherent communication involved but because in this case the communication is overlapped with computation, the associated costs are absent. The inherent communication cost is dependent on task scheduling and load imbalance (which may arise due to either differing processor speeds and/or the work distribution across processors).

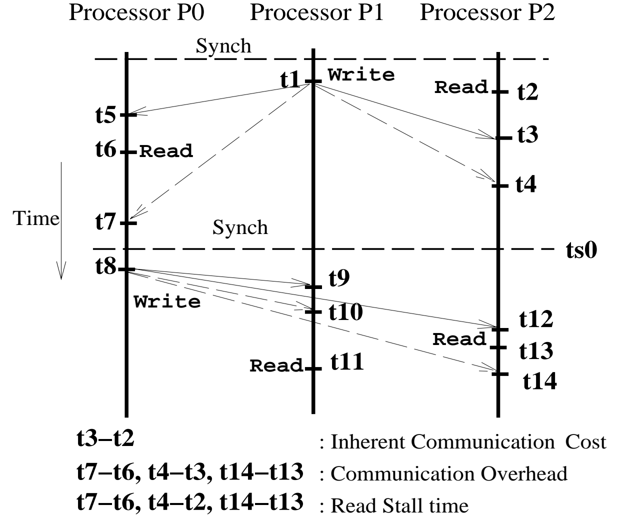

Figure 1: Inherent Communication Costs and Overheads

Any communication costs that arise in addition to this intrinsic requirement of the application is the overhead resulting from a specific shared memory system. For instance in Figure 1, although we said processor $P0$ has no inherent communication costs because of the read at $t6$, due to various architectural factors, the write by $P1$ actually propagates to $P0$ at time $t7$ and the $P0$ sees a cost of ($t7$ - $t6$). In this case the observed penalty is only due to overheads. Latency and contention on the network, the memory consistency model, the cache coherence strategy, and the parameters of the cache subsystem combine to give rise to these overheads. These overheads manifest in different ways depending on the memory system and may be categorized as *read stall* time, *write stall* time, *buffer flush* time. Read stall time is the wait time seen by a typical processor for read-misses; write stall time is the wait time seen by a typical processor for write misses; and buffer flush time is the wait time seen by a typical processor at synchronization points for outstanding writes to complete. In Figure 1 the overhead for the read by processor $P0$ manifests itself as read stall time. Notice that a read stall does not mean that all of it is because of overheads -

some of it may be due to the inherent communication requirements of the application, as in the case for processor $P2$. It should also be clear that if the write by processor $P1$ at time $t1$ stalls the processor from doing useful work in order to provide certain memory consistency guarantees, the time taken for $P0$ to reach synchronization point $ts0$ would be correspondingly longer.

Synchronization events have two cost components. One is for process synchronization to provide guarantees that a task has reached a certain point in execution and/or to coordinate with other tasks. This cost is inherent in the application [2]. The other cost comes about because the architectural choices made require that certain properties hold true at synchronization points. For instance, in the case of weaker memory models, synchronization is the point at which guarantees are made about the propagation of writes. This latter factor which manifests itself as buffer flush time is part of the communication overheads that were introduced by the architecture.

Thus, there is a necessity for a realistic machine model that can predict tighter lower bounds that includes the inherent communication costs to help us understand how close a given system is to the realistic ideal. Such a model can then be used to benchmark different design parameters of a memory system. The knowledge of a realistic ideal that includes communication costs can also serve as a starting point in the design of a zero overhead machine.

2.2 Description of the Model

We want the model to be cognizant of the computation and communication speeds of the underlying architecture. The communication speed helps determine the inherent communication cost of executing a given application on the architecture commensurate with the application's communication pattern; and the compute speed gives us an estimate of how much of this communication is overlapped with computation based on the timing relationships between the producers and consumers of data. Architectural artifacts such as finite caches introduce extra communication caused by replacements (due to the working set not fitting in the cache or due to poor replacement policies) and we treat them as part of the overheads introduced by the memory system. Hence, we do not want the model to be aware of the architectural limits such as finite-sized caches, or limitations of the interconnection network topology. We thus define a *zero overhead* machine (*z-machine*) as follows: it is a machine in which the only communication cost is that necessitated by the pure data flow in the application. In the z-machine, the producer of a datum knows exactly who its consumers are, and ships the datum immediately to these remote consumers. The producer does not wait for the data to reach the consumer and can immediately proceed with its computation. We abstract out the details of the underlying communication subsystem to the point that the datum is available at a consumer after a latency L. The latency is determined only by communication speed of the interconnection network. That is, there is no contention in the z-machine for the data propagation. While this assumption has the obvious benefit of keeping the model simple, it is also realistic since we do not expect the inherent communication requirements in an application to exceed the capabilities of the network. If the consumer accesses the data at least L units of time after it is produced, then the communication is entirely overlapped with computation at both ends, and thus hidden from the application. Clearly, no real memory system can do better than the z-machine in terms of hiding the communication cost of an application. At the same time, it should also be clear that since the z-machine uses the real compute and communication times of the underlying architecture it gives a "realistic" lower bound on the true communication cost in the application. This inherent communication in an application manifests itself as read-stall times on the z-machine.

We should note that our base machine model, the z-machine, is different from base machines or idealized machines usually used in other cache studies [1, 9]. The base model in the other studies typically refers to a machine to which enhancements are applied. Many of those studies consider a sequentially consistent machine using the same cache protocol as the base model. Some other studies idealize the machine by removing some costs (like the protocol processing for coherence messages). While these models may serve the purpose in the respective studies, our model is intended for a different purpose - that of capturing the inherent communication costs and overheads. With the other models it may be possible to see benefits of the protocols in consideration. However, the extent of improvements that further enhancements may be able to obtain, is not always clear.

2.3 Overheads due to Memory Systems

Any real memory system adds overheads not present in the z-machine. For instance, the coherence protocol and the memory consistency model add to these costs in several ways. Since there is no way of pre-determining the consumers of a data item being written to, the memory system has to decide 'when' and 'where' the data should be sent. In update-based protocols, the data is sent to all processors that have currently cached this item. This increases contention on the network (since some of these might be useless updates), and manifest as write-stall times in the execution of the application. In invalidation-based protocols, the 'where' question is handled by not sending the data item to any processor (but rather by sending invalidation messages). In this case all consumer processors incur an access penalty (which manifests as read-stall times in the execution of the application) in its entirety for the remote access even if the data was produced well ahead of the time when it is actually requested by the consumers.

[2] These inherent costs may certainly be changed by restructuring the application, but we are concerned only with overheads given a specific mapping of the application.

Also, the z-machine can be thought of as having an infinite cache at each node. Thus real memory systems incur additional communication due to capacity and conflict misses. Further, if the memory system uses a store buffer to implement a weaker memory model, there could be additional overheads such as buffer-flush times in the execution of the application. Thus by considering each design choice in isolation with the z-machine we can determine the overhead introduced by each hardware artifact on the application performance.

By gradually developing the actual cache subsystem on top of the base machine (and hence introduce the limitations), we can isolate the effects of each of the factors individually.

3 Implementation of the z-machine

The z-machine is not physically realizable. However, we want to incorporate it into an execution-driven simulator so that we can benchmark application performance on different memory systems with reference to the z-machine. For this purpose, we have simulated the z-machine within the SPASM framework [13, 14] an execution-driven parallel architecture simulator. SPASM provides a framework to trap into the simulator on every shared memory read and write. We provide routines that are invoked on these traps based on the machine model we are simulating. The SPASM preprocessor also augments the application code with cycle counting instructions and uses a process oriented simulation package to schedule the events. Further, the simulation kernel provides a choice of network topologies. The simulated shared memory architecture is CC-NUMA using a directory-based cache coherence strategy. The cache line size is assumed to be exactly 4 bytes in the z-machine so that the only communication that occurs is due to true sharing in the application. The z-machine assumes that each producer is an oracle, meaning that the producer knows the set of consumers for a data item. In general, due to dynamic nature of applications we may not be able to determine the recipients of each write. Thus, we simulate the oracle by sending updates to all the processors on writes. The latency for these updates is directly available from the link bandwidth of the network, and is accounted for in the simulation. In order to correctly associate a read with a particular write, a counter is associated with each directory entry. Upon a write, the counter associated with the corresponding memory block is incremented. The counter is decremented after a period of L (the link latency), at which time the updates should be available at all the caches. In order to ensure that a read returns the value of the corresponding write, a read returns a value only if the counter is zero. Otherwise, the read stalls until the counter becomes zero.

The applications we are interested in are data-race free. Thus there are synchronization firewalls that separate producer-consumer relationships for shared data access in the application. The z-machine simulation has to take care of the synchronization ordering in the application. Synchronization is normally used for process control flow; but in memory systems that use weaker memory models it is also used to make guarantees about the program data flow. We take care of this separation in the simulation of the z-machine as follows. The synchronization events simulate only the process control flow aspects. Not flushing the buffers at synchronization points implies that we cannot guarantee a correct value to be be available at other processors after the synchronization point. However, the counter mechanism that we described earlier is used to delay consumers if a produced value has not been propagated. This mechanism is sufficient to ensure that consumers receive the correct value, and we thus free the synchronization events from providing data flow guarantees required by the weak memory models. It should be noted that the notion of memory consistency implemented in the z-machine is the weakest possible consistency commensurate with the data access pattern of the application.

4 Memory Systems

As described earlier, the base hardware is a CC-NUMA machine. Each node has a piece of the shared memory with its associated full-mapped directory information, a private cache, and a write buffer (not unlike the Dash multiprocessor [10]). In our work we consider the following four memory systems (RCinv, RCupd, RCadapt, RCcomp) that are built on top of the base hardware by specifying a particular coherence protocol along with a memory model.

RCinv: The memory system uses the release consistent (RC) memory model [6] and a Berkeley-style write-invalidate protocol. In this system, a processor write that misses in the cache is simply recorded in the write-buffer and the processor continues execution without stalling. The write is completed when ownership for the block is obtained from the directory controller, at which point the write request is retired from the buffer. In addition to the read stalls, a processor may incur write stalls if the write-buffer is full upon a write-miss, and incurs a buffer flush penalty if the buffer is non-empty at a release point.

RCupd: This memory system uses the RC memory model, a simple write-update protocol similar to the one used in the Firefly multiprocessor [15]. From the point of view of the processor, writes are handled exactly as in RCinv. However, we expect a higher write stall time for this memory system compared to RCinv due to the larger number of messages warranted by update schemes. To reduce the number of messages on the network we assume an additional *merge buffer* at each node that combines writes to the same cache line. While it has been shown that the merge buffer is effective in reducing the number of messages [5], it does introduce additional stall time for flushing the merge buffer at synchronization points for guaranteeing the correctness of the protocol.

RCcomp: This memory system also uses the RC mem-

ory model, and a merge buffer as in RCupd. However the cache protocol used is slightly different. Simple update protocols are incapable of accommodating the changing sharing pattern in an application, and thus are expected to perform poorly. The redundant updates incurred in such protocols increase both the stall times at the sender as well as potentially slowing down other unrelated memory accesses due to the contention on the network. The RCcomp memory system uses a competitive update protocol to alleviate this problem. A processor self-invalidates a line that has been updated *threshold* times without an intervening read by that processor. By decreasing the number of messages this memory system is expected to have lower write stall and buffer flush times when compared to RCupd.

RCadapt: This memory system is also based on the RC memory model but uses an adaptive protocol that switches between invalidation and update based on changes in the sharing pattern of the application. The protocol used in this memory system was developed for software management of coherent caches through the use of explicit communication primitives [11]. The directory controller keeps state information for sending updates to the active set of sharers through the *selective-write* primitive. When the selective-write primitive is used the corresponding memory block enters a special state. The presence bits in the directory represent active set the of sharers of that phase. Whenever a processor attempts to read a memory block with an already established sharing pattern (which implies the memory block is in a special state) it is taken as an indication of a change in the sharing pattern with the application entering a new phase. Therefore the directory controller re-initializes (by invalidating the current set of sharers) through an appropriate state transition upon such reads. In this study, we treat all writes to shared data as selective-writes. This memory system is expected to have read stall times comparable to update based schemes, and write stall times comparable to invalidate schemes.

5 Performance Results

In this section, we use four different applications to evaluate the memory systems presented in the previous section. In most memory systems studies, a sequentially consistent invalidation-based protocol is used as the frame of reference for benchmarking the performance of a given memory system. While this is a reasonable approach for benchmarking a given memory system, it fails to give an idea of how far off its performance is from a realistic ideal. The objective of our performance study is to give a break down of the overheads introduced by each memory system relative to such a realistic ideal which is represented by the z-machine. Therefore, we limit our discussions to executions observed on a 16 node simulated machine. For all the memory systems, we assume the following: infinite cache size; a cache block size of 32 bytes (as mentioned earlier, in the case of the z-machine the cache block is 4 bytes to discount overheads and effects caused by larger

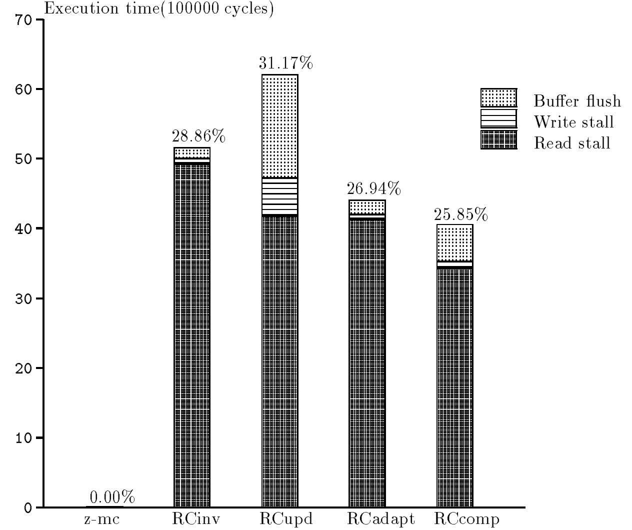

Figure 2: Cholesky

line sizes); a mesh interconnect with a link latency of 1.6 CPU cycles per byte; store buffer of size 4 entries; and a merge buffer of one cache block. The applications we studied include Cholesky and Barnes-Hut from SPLASH suite [12], Integer Sort from the NAS parallel benchmark suite [3], and Maxflow [2].

Cholesky performs a factorization of a sparse positive definite matrix. The sparse nature of the matrix results in an algorithm with a data dependent access pattern. Sets of columns having similar non-zero structure are combined into supernodes. Supernodes are added to a central work queue if a set of criteria are satisfied. Processors get tasks from this central work queue. Each of these tasks is used to modify subsequent supernodes and if the criteria of the supernode being changed are satisfied then that node is also added to the work queue. Communication is involved in fetching all the required columns to a processor working on a given task and to obtain tasks from the central work queue. The communication pattern is totally dynamic based on the access to the work queue. In the study we consider a 1086x1086 matrix with 30,824 floating point non-zeros in the matrix and 110,461 in the factor with 506 supernodes. Barnes-Hut is an N-body simulation application. The application simulates over time the movement of these bodies due to the gravitational forces exerted on one another, given some set of initial conditions. The implementation statically assigns a set of bodies to each processor and goes through three phases for each time step. The producer-consumer relationship is well-defined and gradually changes over time. The problem size considered in this study used 128 bodies over 50 time steps [3]. The integer sort kernel uses a parallel bucket sort to rank a list of integers. The communication pattern

[3]The problem included an artificial boost to affect the sharing pattern every 10 time steps. This boost simulates the effects of more time steps.

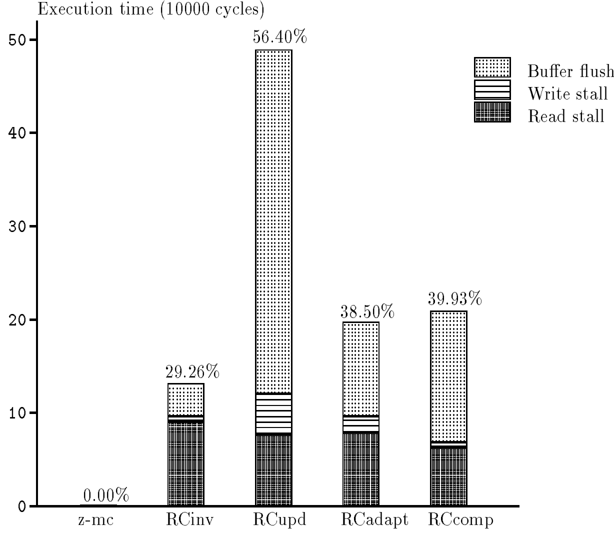

Figure 3: IS

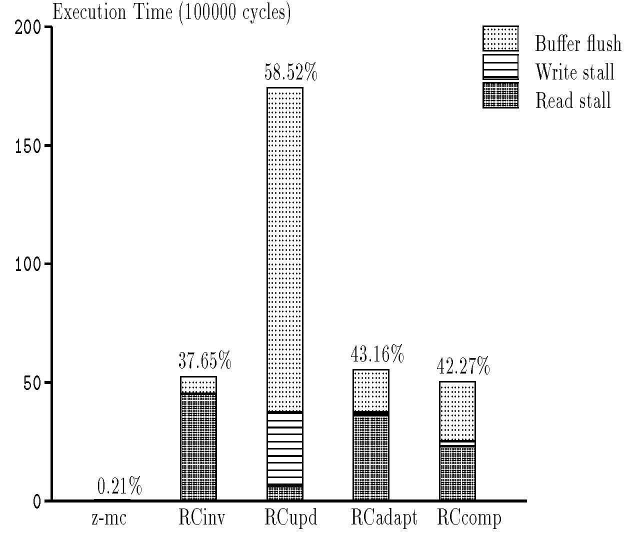

Figure 4: Maxflow

is well defined statically. The problem size considered is 32K integers with 1K buckets. The Maxflow application finds the maximum flow from the distinguished source to the sink, in a directed graph with edge capacities. In the implementation [2], each processor accesses a local work queue for tasks to perform. These may in turn generate new tasks which are added to this local work queue. Each task involves read and write accesses to shared data. The local queues of all processors interact via a global queue for load balancing. Thus the producer-consumer relationship for data in this application is quite dynamic and random. The amount of computation required in the application is small and most of the time is spent in data movement. The input graph we consider has 200 vertices and 400 bidirectional edges.

As we mentioned earlier (see Section 2), by definition the z-machine will not have any write stall or buffer flush times. The inherent communication cost in the application will manifest as read stall times due to the timing relationships for the memory accesses from the producers and consumers of a data item. From our simulation results we observe that this cost is virtually zero for all the applications (see Figures 2,3,4 and 5). This indicates that all the inherent communication cost can be overlapped with the computation. Table 1 gives the time spent by each application on the network in the z-machine, most of which is hidden by the computation. This is a surprising result since the z-machine does take into account the two important parameters of any parallel machine, namely, the compute and the communication speeds in accounting for the inherent communication costs. In other words, the performance on the z-machine for these applications matches what would be observed on a PRAM. Given this result, any communication cost observed on the memory systems is an overhead.

Figures 2,3,4 and 5 show the performance of these applications on the memory systems discussed in the previous section. The y-axis shows the execution time. For each memory system we have a bar which shows the amount of time due to each of the overheads. The total percentage of the overall execution time that these overheads represent is shown on the top each bar. The figures show that the overheads range from 6% to 37% for RCinv, 3% to 58% for RCupd, 3% to 42% for RCcomp, and 3% to 43% for RCadapt for the applications considered. As expected, the dominant component of the overheads for RCinv is the read stall time, and is significantly higher than those observed for the other three memory systems. The difference between the read stall observed on RCupd and the z-machine gives the read stall due to cold misses for any memory system[4](since there are no capacity of conflict misses with the infinite cache assumption). Significant difference in the read stall times between RCinv and RCupd implies data reuse. This is true for Barnes-Hut and Maxflow applications (see Figures 5,4), and not true for Cholesky and IS (see Figures 2,3). The RCcomp and RCadapt can exploit data reuse only when the application has well established producer-consumer relationships

Application	Number of Writes	% of Total Execution Time that these writes represent	Observed Costs (in cycles)
Cholesky	103915	1.477	54.6
IS	6353	3.779	0.0
Maxflow	38209	1.788	7257.8
Nbody	4542	0.002	0.0

Table 1: Inherent communication and observed costs on the z-machine

[4]Note that the observed read stall time due to cold misses could be slightly higher in an update protocol due to increased contention owing to update traffic.

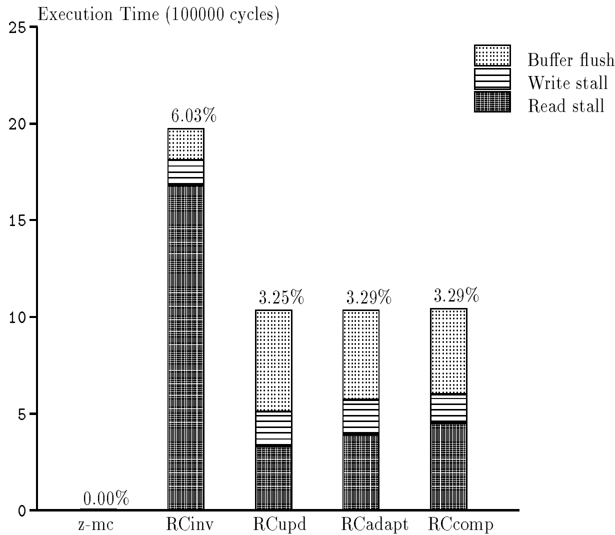

Figure 5: Barnes-Hut

over a period of time. This can be seen in the read stall times observed for Barnes-Hut in Figure 5. On the other hand, in Maxflow (see Figure 4) the producer-consumer relationship is more random making the read stall times for RCcomp and RCadapt to be similar to that of RCinv.

As expected, the write stall times for RCinv are significantly lower when compared to the other three. The write stall time is directly related to the number of messages that a protocol incurs on the network. Due to the dynamic nature of RCadapt and RCcomp (in switching between invalidation and update), these two memory systems incur lesser number of messages than RCupd. Decreasing the number of messages frees up the store buffer sooner, and also reduces the contention on the network.

The buffer flush time is a function of the store buffer size, the merge buffer size, and the frequency of synchronization in the application. The use of merge buffer results in a significant increase of buffer flush time for RCupd, RCcomp, and RCadapt compared to RCinv. Contention on the network plays a major role on the buffer flush time. This is evident when we compare RCupd with RCadapt and RCcomp.

6 Architectural Implications

The first observation is that since the z-machine has close to zero read stall times, the corresponding times observed on RCinv are unwarranted. The observation implies that it is possible to overlap most of the communication with the computation in an application. Figure 1 illustrates the above situation. The write by processor $P0$ at $t8$ is propagated well before the intended read by processor $P1$ at $t11$ even including the overheads. Our goal should be to reduce the unwarranted costs without burdening the application programmer. One approach to reach the goal is moving towards an update based protocol, where this overhead component is usually low and is due to cold misses. However update based protocols do not handle all the situations – our results showed large read stall times even in the update protocol for Cholesky. Thus, another approach is to employ effective prefetching strategies. Applications in which there is considerable cold miss penalty (for e.g. Cholesky) prefetching and/or multithreading are more promising options.

We know that the write stall and buffer flush times are pure overheads from the point of view of an application. In order to realize a zero overhead machine we have to drive these two components to zero. The excessive message traffic of the update protocols is the main culprit for these components. To keep the traffic low (and thus the overheads), it is important for the protocol to adapt to changing sharing patterns in the applications (i.e. lean towards protocols such as RCadapt and RCcomp). Such an adaptive protocol could bring these two overhead components close to what is observed for RCinv. In our quest for a zero overhead machine, we now have to identify architectural enhancements that will drive this overhead further down. Write stall time is dependent on two parameters: the store buffer size and the relative speed of the network with respect to the processor. Improving either of these two parameters will help to lower the write stall time.

Increasing the write buffer size could potentially increase the buffer flush time. The buffer flush time is a result of the RC memory model that links the data flow in the program with the synchronization operations in the program. As the performance on the z-machine indicates, there is an advantage in decoupling the two, i. e., use synchronization only for control flow and use a different mechanism for data flow. The motivation for doing this is to eliminate the buffer flush time. One approach would be associating data with synchronization [7] in order to carry out smart self-invalidations when needed at the consumer instead of stalling at the producer.

7 Concluding Remarks

The goal of several parallel computing research projects is to realize scalable shared memory machines. Essentially, this goal translates to making a parallel machine appear as a zero overhead machine from the point of view of an application, In striving towards this goal, we first need a frame of reference for the inherent communication cost in an application. We developed a realistic machine model the z-machine which would help to serve as such a frame of reference. We have incorporated the z-machine in an execution-driven simulator so that the inherent communication cost of an application can be quantified. An important result is that the performance on the z-machine for the applications used in this study matches what would be observed on a PRAM. Using the performance on the z-machine as a realistic ideal to strive for, we benchmarked four different memory systems. We presented a break-

down of the overheads seen in these memory systems and derived architectural implications to drive down these overheads.

There are several open issues to be explored including the effect of finite caches on the overheads, the use of other architectural enhancements such as multithreading and prefetching to lower the overheads, and the design of primitives that better exploit the synchronization information in the applications.

Acknowledgment: The selective-write primitive used in the adaptive protocol was co-designed with Anand Sivasubramaniam and Dr. H. Venkateswaran. We would also like to thank the members of our architecture group meetings, particularly Dr. H. Venkateswaran, Vibby Gottemukkala, Anand Sivasubramaniam and Ivan Yanasak, with whom we had many helpful discussions.

References

[1] F. Dahlgren aand M. Dubois and P. Stenstrom. Combined performance gains of simple cache protocol extensions. In *The 21st annual international symposium on computer architecture*, pages 187–197, April 1994.

[2] R. J. Anderson and J. C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 168–77, June 1992.

[3] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[4] M. E. Crovella and T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing '94*, November 1994.

[5] F. Dahlgren and P. Stenstrom. Reducing the write traffic for a hybrid cache protocol. In *1994 International Conference on Parallel Processing*, volume 1, pages 166–173, August 1994.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. 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.

[7] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[8] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[9] M. Heinrich, J. Kuskin, D. Ofelt, and et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[10] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *Transactions on parallel and distributed systems*, 4(1):41–61, January 1993.

[11] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[12] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[13] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[14] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 22(3):411–426, September 1994.

[15] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. In *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[16] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.