The Quest for a Zero Overhead Shared Memory Parallel Machine*

Gautam Shah
Aman Singla
Umakishore Ramachandran
Technical Report GIT-CC-95-06
January 1995

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Phone: (404) 894-5136
Fax: (404) 894-9442
e-mail: 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 is the performance of a given memory system from a realistic ideal parallel machine. We define such a realistic machine model called the *z-machine*, which 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 to be used as a reference for benchmarking for 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.

Key Words:
Shared memory multiprocessors, Execution-driven simulation, Communication overheads, latency tolerating techniques, synchronization, Cache coherence protocols,

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

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. Several techniques including relaxed memory consistency models, coherent caches, explicit communication primitives, and multithreading have been proposed to reduce and/or tolerate the 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 generally recognized that no one technique is universally applicable for reducing or tolerating the communication overheads in all situations [6].

There have been several recent studies in separating the overheads seen in the execution of an application on a parallel architecture [3, 10]. 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. [10] 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 a 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 that 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 which has zero communication overhead from the point of view of an application is the goal of this work. PRAM [13] has been used quite successfully as a vehicle for parallel algorithm design. In [10], it is shown how PRAM could be used 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 is too unrealistic for performance estimation since the model assigns unit cost for all memory accesses. Therefore, it is impossible to get the communication that is inherent in the mapping of an application onto a given architecture.

We develop a *base machine model* that achieves this objective of quantifying the inherent communication in an application (Section 2). 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 by 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. An equally important contribution is the breakdown of the overheads seen in the different memory systems in the context of four different applications.

2 Base Machine Model

2.1 Communication in an Application

Before we identify the overheads that may be introduced by the 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. The synchronization operations used in the application are just firewalls to ensure that the program is data-race free given this producer-consumer relationships. Depending on the control flow of the application some of this communication may be statically determinable and others dynamic. Depending on the latency for remote communication on the interconnection network, these requirements will translate a certain *lower bound* on the communication costs that is inherent in the application. In striving for a base machine model, our goal is to be able to quantify this communication cost.

Any communication that takes place in addition to this intrinsic requirement of the application is the overhead resulting from a specific shared memory system. 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. 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. 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.

Thus there is a necessity for a realistic machine model that can predict tighter lower bounds including 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.

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 will help determine the inherent communication cost of executing a given application on this architecture commensurate with the application's communication pattern; and the compute speed will give 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. We do not want the model to be aware of any other architectural limits such as finite-sized caches, or 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. The datum is available at a consumer after a latency L determined only by the communication speed of the underlying architecture. 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 not unrealistic 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 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. It should also be clear that all the communication cost seen in the z-machine is due entirely to read-stalls.

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 multiple 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 encached this item. This will increase 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 will incur an access penalty (which will manifest 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. Also, the z-machine can be thought of as having an infinite cache at each node. Thus real memory systems will 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 which manifest 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 [10, 11] an execution-driven parallel architecture simulator. The simulated shared memory architecture is CC-NUMA using a directory-based cache coherence strategy. The z-machine assumes that each producer knows the set of consumers for a data item. This is simulated 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. A counter is associated with each directory entry. Upon a write, the counter associated with this memory block is incremented. The counter is reset after a period of L (the link latency), at which time the updates should be available at all the caches. A read returns a value only if the counter is zero.

Otherwise, it has to block. The cache line size is assumed to be exactly 4 bytes in the z-machine so that the only communication that is seen are due to true sharing in the application.

It is assumed that 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 for making guarantees about the program data flow. In the simulation of the z-machine the data flow is controlled by the counter mechanism, and thus the synchronization in the application is used only for process control flow. It should be noted that the notion of memory consistency implemented in the z-machine is the weakest possible commensurate with the data access pattern of the application.

4 Memory Systems

As has been 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 [7]). In our work we consider the following four (RCinv, RCupd, RCadapt, RCcomp) memory systems 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 [5] 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 incur a buffer flush penalty if the buffer is non-empty at a release point.

RCupd: This memory system uses RC memory model, a simple write-update protocol similar to the one used in the Firefly multiprocessor [12]. 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 [4] that the merge buffer is effective in reducing the number of messages, 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 memory 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 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 [8]. The directory controller keeps state information for sending updates to the active set of sharers through the *selective-write* primitive. Whenever a processor attempts to read a memory block with an already established sharing pattern 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

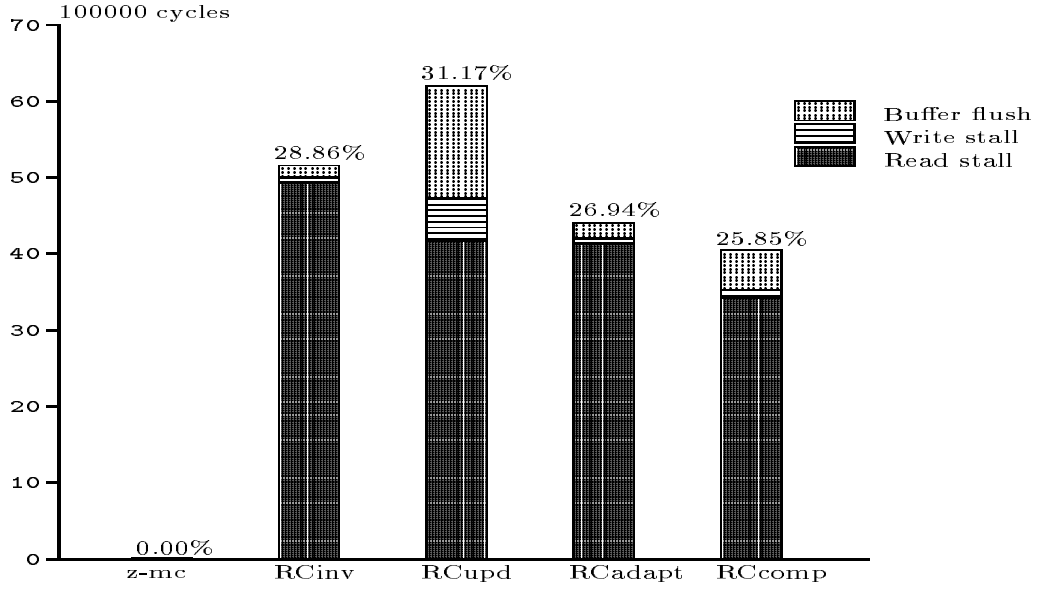

Figure 1: Cholesky

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 is its performance 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; 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 Nbody from SPLASH

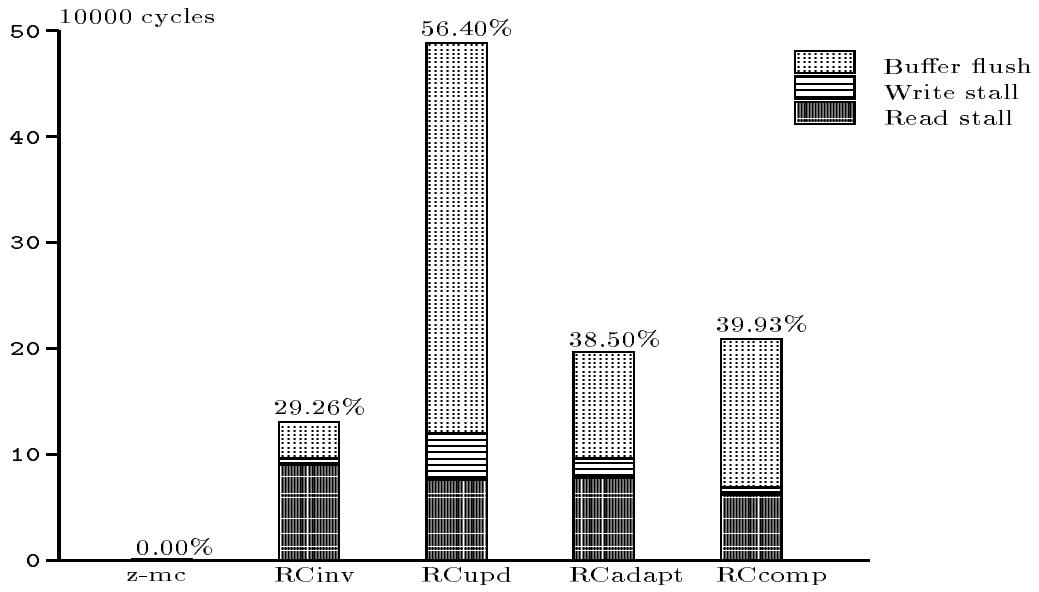

Figure 2: IS

suite [9], Integer Sort from the NAS parallel benchmark suite [2], and Maxflow [1].

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 1,2,3 and 4). 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.

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

Figures 1,2,3 and 4 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

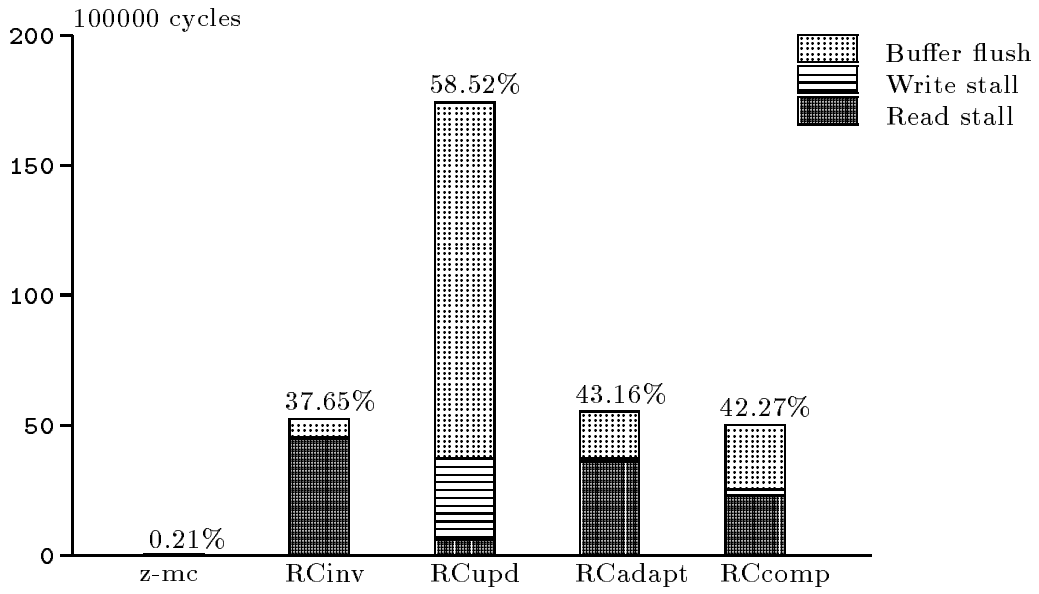

Figure 3: Maxflow

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[1](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 Nbody and Maxflow applications (see Figures 4,3), and not true for Cholesky and IS (see Figures 1,2). The RCcomp and RCadapt can exploit data reuse only when the application has well established producer-consumer relationships over a period of time. This can be seen in the read stall times observed for Nbody in Figure 4. On the other hand, in Maxflow (see Figure 3) the producer-consumer relationship is more random making the read stall times for RCcomp and RCadapt to be similar to that of RCinv.

The write stall times are significantly lower for RCinv compared to the other three as expected. 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.

[1]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.

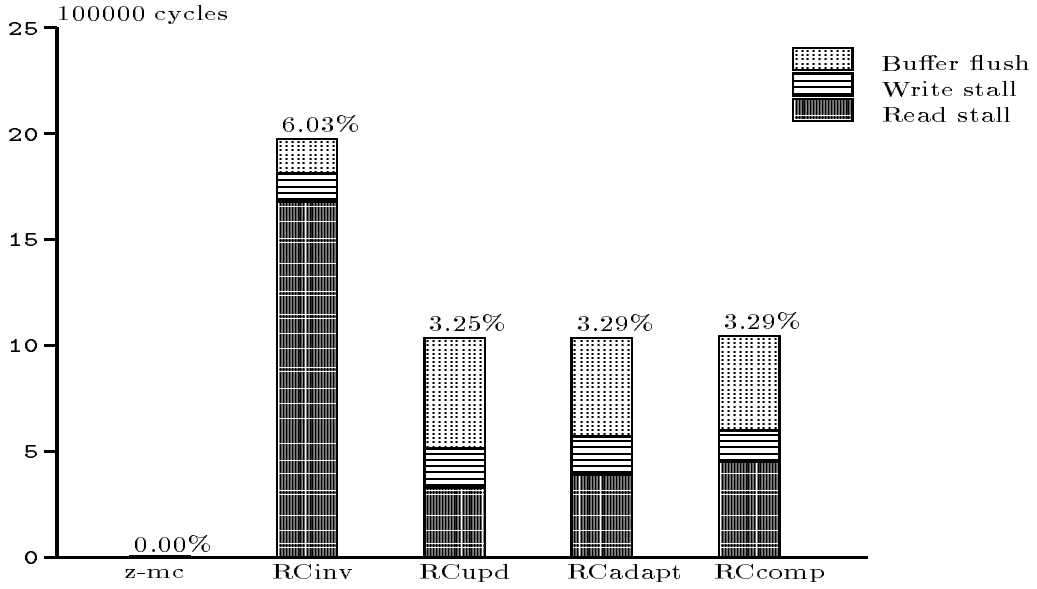

Figure 4: Nbody

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. One approach is moving towards an update based protocol, where this overhead component usually low and is due to cold misses. 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. To achieve 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 figure out architectural enhancements that will drive this overhead further down. Write stall time is 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 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 breakdown 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.

References

[1] 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.

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

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

[4] F. Dahlgren and P. Stenstrom. Using write caches to improve performance of cache coherence protocols in shared memory multiprocessors. Technical report, Dept. of Comp. Eng., Lund Univ., April 1993.

[5] 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.

[6] 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.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[8] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. Technical Report GIT-CC-94/59, College of Computing, Georgia Institute of Technology, December 1994.

[9] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[10] 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.

[11] 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.

[12] 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.

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