Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors

Olivier C. Maquelin[1], Herbert H.J. Hum[2], Guang R. Gao[1]

[1] McGill University, School of Computer Science
3480 University St., Montréal, Canada, H3A 2A7
[2] Concordia University, Dept. of Electrical and Computer Engineering
1455 de Maisonneuve W., Montréal, Canada, H3G 1M8

Abstract. Multithreaded architectures have been proposed for future multiprocessor systems due to their ability to cope with network and synchronization latencies. Some of these architectures depart significantly from current RISC processor designs, while others retain most of the RISC core unchanged. However, in light of the very low cost and excellent performance of off-the-shelf microprocessors it seems important to determine whether it is possible to build efficient multithreaded machines based on unmodified RISC processors, or if such an approach faces inherent limitations. This paper describes the costs and benefits of running multithreaded programs on the EARTH-MANNA system, which uses two Intel i860 XP microprocessors per node.

1 Introduction

Multithreaded architectures [1, 2, 12] have been promoted as potential processing nodes for future parallel systems due to their inherent ability to tolerate network and synchronization latencies. These delays are hidden by letting the processing unit switch to a different thread of execution instead of idling until the operation has completed. Due to the additional synchronization overhead when taking advantage of parallelism at a finer level, many architects question whether multithreading support can be made transparent to sequentially executing code and still be useful. However, preliminary results gained with the EARTH-MANNA system show that multithreading can indeed be useful, even on machines built with conventional RISC microprocessors. Even though the Intel i860XP processor used in EARTH-MANNA was not designed for multithreading, benchmark results show that good speedups can be achieved, even compared with an efficient sequential implementation. Moreover, a detailed analysis of the multithreading overheads shows that they could be reduced significantly without having to switch to a custom processor design.

1.1 The EARTH-MANNA system

The results discussed in this paper were gained with our implementation of the EARTH *(Efficient Architecture for Running THreads)* model [4, 6, 5] on top of

the *MANNA* (Massively parallel Architecture for Non-numerical and Numerical Applications) multiprocessor [3] developed at GMD-FIRST in Berlin, Germany. Each node of a MANNA machine consists of two Intel i860XP RISC CPUs, clocked at 50 MHz, 32 MB of dynamic RAM and a bidirectional network interface capable of transferring 50 MB/s in each direction. This dual-processor design is similar to the EARTH model (see Fig. 1), which separates the processing node into an *Execution Unit* (EU) and a *Synchronization Unit* (SU).

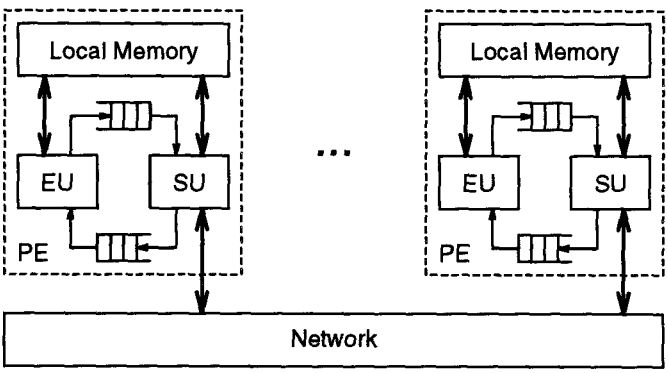

Fig. 1. The EARTH architecture

As demonstrated in [5], it is possible to implement multithreading support for such a machine without a major impact on performance. Performance of the parallelized code on a single node can be close to that of the sequential code because substantial portions of the code can often be executed in the normal sequential way. As shown in Sect. 3.3, performance gains can also be achieved by taking advantage of the SU to off-load data transfers from the main CPU. It is also interesting to note that for the EARTH-MANNA system the cost of saving and restoring registers is only a relatively small fraction of the total context switch costs (see Sect. 2.4). This means that better hardware support for multithreading should focus primarily on reducing the remaining costs, which are mostly due to communication between the EU and SU.

1.2 Synopsis

The next section discusses the overall performance of the EARTH-MANNA system and the costs associated with the multithreading support, such as the overhead to issue a split-phase transaction and the context switching costs. Then, Sect. 3 gives some insights into the relative performance of single-processor vs. dual-processor node designs. The costs for all internal operations involved in a remote memory access are shown for both cases, and finally some experimental results showing the benefit of a second processor are discussed.

2 Performance of the Multithreading Support

This section discusses the performance of the EARTH-MANNA multithreading support. It first shows the performance of some typical operations, then goes into more details to describe the EU overhead. The cost of communication between EU and SU is shown to be significant, mainly due to the necessary DRAM accesses. In contrast, the costs due to function invocation and to saving or restoring registers are shown to be relatively modest.

2.1 Performance of Typical EARTH Operations

In order to exploit parallelism at a finer level of granularity, it is important to reduce the costs of small messages, such as remote function calls or single-word remote memory accesses. While more conventional architectures focus primarily on high network bandwidth, the EARTH-MANNA system also tries to achieve very low latencies for the most important operations (see Table 1).

EARTH Operation	*Sequential Local*	*Sequential Remote*	*Pipelined Local*	*Pipelined Remote*
Spawn thread	2354 ns	4286 ns	2003 ns	1570 ns
Load word	2648 ns	7109 ns	1137 ns	1908 ns
Store word	2560 ns	6458 ns	1060 ns	1749 ns
Function call	5296 ns	9216 ns	3188 ns	2792 ns

Table 1. Execution time of some EARTH operations

These are overall costs, which include the network delays and the synchronization overhead. A substantial part of these costs can be hidden through multithreading. The actual costs to the EU will be described in more detail in Sect. 2.2. Because the EARTH model implements split-phase transactions, two or more EARTH operations are necessary to implement the operations in Table 1. See Sect. 3.1 for an example showing the detailed costs for a remote load.

Four numbers are shown in Table 1, each corresponding to a different usage of the operation. The typical latency depends on the location of the data that is referenced or the thread that is started. Remote references necessitate network accesses, which slows down execution. *Sequential* and *Pipelined* are two extremes of the typical usage. The sequential value indicates how long it takes to perform the complete operation, including context switching. In the pipelined case, on the other hand, operations are issued as fast as possible, without the need to synchronize before issuing the next operation. Obviously, the pipelined numbers are lower, as the EU, SU and network can all work in parallel.

These numbers compare well with other architectures, even those with hardware support for remote memory accesses. For example, in the Stanford DASH

multiprocessor [8] a remote load takes 3 μs. In the Stanford FLASH architecture [7], a remote load will be performed in about 1 μs. However, the processor and network speed of the FLASH architecture are about 4 – 6 times faster than the corresponding numbers in the MANNA architecture. The relative speed of communication vs. computation will therefore not be very different.

Performance of bulk data transfers is also excellent, with a maximum bandwidth of 41 MB/s in one direction and 61 MB/s when data is transferred in both directions simultaneously. The limiting factor in the first case is the packetization overhead, which reduces the usable link bandwidth by nearly 20%. In the second case the i860XP's memory interface becomes the bottleneck. These values are especially good considering that our fastest local memory to local memory copy routine achieves 77 MB/s.

2.2 EU Overhead for the EARTH Operations

The execution times shown in Table 1 consider the total execution time. However, because the EU and SU can work in parallel, the EU can switch to a different thread while the SU performs the actual work. Table 2 shows the remaining EU overhead for some typical EARTH operation, as well as the time needed by the local SU and the remote SU to do the actual work. The EU overhead is the only part of the execution costs that can not be masked by multithreading. A more complete table can be found in [5].

EARTH Operation	*Local EU Overhead (ns)*	*Local SU Costs (ns)*	*Remote SU Costs (ns)*
SYNC (local)	700	200	0
SYNC (remote)	700	500	700
GET_SYNC (local)	780	300	0
GET_SYNC (remote)	780	500	800
DATA_SYNC (local)	780	300	0
DATA_SYNC (remote)	780	500	800
INVOKE (local)	800–1500	300–1000	0
INVOKE (remote)	800–1500	500–2000	500–2000
END_FUNCTION	750	0	0
END_THREAD	720	0	0

Table 2. Costs of some EARTH operations

As an example, a GET_SYNC (read word) operation where the data happens to be local costs the EU 780 ns on the average, while it takes the local SU only about 300 ns to perform the actual operation. These numbers are only averages, as the actual cost depends on many factors, such as cache hit rate and bus contention. Also, the EU overhead includes the EU $\leftrightarrow$ SU communication costs, but these costs are not included in the SU numbers.

Again, these numbers compare well with similar architectures. For example, with Active Messages on the CM-5 [13] the overhead to send a message is 1.6 μs. However, because communication is not off-loaded to a second processor, the overhead of receiving that message (1.7 μs) and executing the corresponding handler also has to be taken into account. The costs for the complete operation are therefore significantly larger than the typical 700 – 800 ns EU overhead of the EARTH-MANNA system. The J-Machine [10] achieves better performance, due to its hardware support for communication and synchronization. However, it still takes 0.9 μs to send and receive an Active Message handler on that machine. Moreover, handlers are executed on the same processor, thus further slowing down the computation.

Considering that on MANNA each off-chip access takes about 200 – 300 ns, the run-time system is quite efficient. For such low-level code the number of instructions alone is not sufficient to get a good approximation of the execution time. Because communication between the EU and SU goes through software queues in the local memory, and because this information is shared between both processors, a certain number of cache misses and invalidations can not be avoided. For example, Fig. 2 shows the code for an END_THREAD operation (which switches to the next ready thread).

```
1:      ld.l 16(r14), r1          // load instruction pointer
        ld.l 20(r14), fp          // load frame pointer
        bte 0, r1, 1b             // branch back if queue empty
        st.l r0, 16(r14)          // mark element empty
        bri r1                    // branch to specified address
        ld.l 0(r14), r14          // fetch address of next element
```

Fig. 2. Implementation of END_THREAD

The processor first loads the new instruction and frame pointers, branching back if the queue element turned out to be empty. After successfully loading the new pointers it marks the element empty, branches to the thread address and moves the queue pointer to the next element. This instruction sequence is quite short and little can be done to make it more efficient without additional hardware support. However, it takes on the average 720 ns to execute, which is significantly more than the 140 ns it would ideally take to execute these 6 instructions (*bri* needs two cycles to execute, for a total of 7 cycles).

The reason for the difference is that this code always involves at least one cache miss and one off-chip write, as shared data has to be transferred from the cache of the first processor to the cache of the second processor and consumed elements have to be freed. Bus conflicts also slow down communication, as the EU has to wait for the SU to release the bus. With today's RISC processors, maximum performance can only be achieved with good cache hit rates, or in

other words when no off-chip accesses are performed. However, communication with another processor necessarily causes off-chip accesses. This turns out to be a crucial factor when using two CPUs per node.

With only the conventional, bus-based communication mechanisms, little can be done to improve performance. However, it is quite possible to improve hardware support for EU $\leftrightarrow$ SU communication while still using an off-the-shelf processor as execution unit. This could be done e.g. with hardware queues that bypass the memory hierarchy (see Sect. 3.2). With adequate support it should be possible to lower the EU overhead for sending a request to around 100 – 200 ns, which would be less than the typical cost of a cache miss.

2.3 Function Invocation

The EARTH-MANNA system distinguishes between 4 different types of function invocations: normal sequential call, sequential call of a threaded function, invocation on a specific remote node, and invocation on an arbitrary node through dynamic load balancing. The frame of a *threaded* function, i.e. a function that contains threads or that calls other threaded functions, is allocated dynamically from the heap. Such functions can then invoke other threaded functions, or they can call sequential functions with the normal stack-based mechanism.

Frame allocation from the heap is necessary for threaded functions, as they can run in parallel with each other and therefore terminate in an arbitrary order. By keeping free lists for the most common frame sizes (i.e. the smallest frame sizes), the typical overhead for dynamic frame allocation can be kept down to around 200 ns. Because normal function calls use the sequential, stack-based calling mechanism, such functions run at the same speed as in a sequential implementation, which is crucial for good overall performance. This also allows the standard system libraries to be used unchanged.

2.4 Context Switching

Typical context switching implementations on conventional machines have to save and restore a large number of processor registers. However, the set of registers that are really active at context switch points, i.e. that really must be saved or restored, is often much smaller. This fact also motivated other architects to develop new mechanisms to speed up context switches, such as e.g. the Named-State Register File [11].

In the EARTH-MANNA system, context switches among threads are explicit and known to the compiler. The EARTH Threaded-C compiler can therefore analyze register usage at thread boundaries and minimize the number of registers saved and restored. The resulting save set is in general a conservative estimate of the registers that are really active, due to the presence of conditional branches and due to other simplifying assumptions made by the compiler. However, even with these limitations it turns out that on the average only a few registers need to be saved or restored. Table 3 shows context switching information that was

gathered from five widely different benchmarks. See [5] for a detailed performance evaluation of these benchmarks.

Benchmarks	*Context Switches*	*Total Saved*	*Total Restored*	*Average Saved*	*Average Restored*
Ray Tracing	258	518	516	2.01	2.00
Protein Folding	65493	150327	177128	2.30	2.70
Paraffins	294	287	2	0.98	0.01
Tomcatv	77776	27131	154123	0.35	1.98
N-Queens	3780338	9868155	9868155	2.61	2.61

Table 3. Registers saved and restored

The average number of registers saved and restored is quite low for all five programs. This seems to be more or less independent of the average thread run-length, which ranges from 14 μs for N-Queens to about 1 s for Ray Tracing. These numbers are specific to the code generator we are using (the EARTH Threaded-C compiler is based on a commercial C compiler from The Portland Group Inc.), as different compilers will keep different variables in registers. However, a closer look at the generated object code did not reveal any serious shortcomings in register usage, but rather reinforced our conviction that the number of active registers at context switch points is often quite low.

Moreover, our experiences showed that in most cases good cache hit rates can be achieved for the loading and saving of registers. This means that the cost per load or store is only about 20 ns. On EARTH-MANNA the save/restore overhead is therefore only a small fraction of the total thread switching costs. It seems therefore much more important to support the EU $\leftrightarrow$ SU communication in hardware than to add support for multiple register sets.

3 Single Processor vs. Dual Processors: a Case Study

This section discusses the benefits of using a dual-processor node design in more detail. A detailed breakdown of costs for the remote load operation gives some insights into what could be expected from a single-processor implementation and what could be gained from better hardware support. Nevertheless, we then show that even without such hardware support some remote memory accesses can be masked by taking advantage of multithreading and a second processor.

3.1 Detailed Execution Costs

To better understand the advantage of having a second processing unit, it is first necessary to understand how much time is spent in each part of a typical operation. Table 4 shows a detailed breakdown of costs for a remote load.

Operation	*Delay*
EU send request to SU	780 ns
SU fetch request from EU	500 ns
SU generate and send message	500 ns
Transmission and polling delay	1000 ns
Remote SU read message from link	500 ns
Remote SU perform GET operation	300 ns
Remote SU generate and send response	500 ns
Transmission and polling delay	1000 ns
SU read response from link	500 ns
SU store result and sync	300 ns
SU insert into ready queue	500 ns
EU fetch next thread information	720 ns
Total	7100 ns

Table 4. Breakdown of costs for the remote load operation

In order to offer the same functionality in a single-processor node, some sort of interrupt (or other periodic polling) mechanism has to be implemented. Table 5 shows the estimated breakdown of costs for an interrupt-driven implementation.

Operation	*Delay*
EU generate and send message	800 ns
Transmission delay	500 ns
Remote EU interrupt latency	1000 ns
Remote EU read message from link	500 ns
Remote EU perform GET operation	300 ns
Remote EU generate and send response	800 ns
Transmission delay	500 ns
EU interrupt latency	1000 ns
EU read response from link	500 ns
EU store result and sync	200 ns
EU fetch next thread information	200 ns
Total	6300 ns

Table 5. Costs for an interrupt-driven implementation of remote loads

These numbers are estimates, as no interrupt-driven version of the run-time system has been implemented yet. For example, it is difficult to exactly predict the costs of interrupt latencies. A rather optimistic value was chosen for that delay to make sure that our comparison is not unfair to the interrupt-driven version.

In the single-processor, i.e. the interrupt-driven implementation, it is not necessary to send messages through the second processor; thus the total elapsed time for a remote load operation is smaller (6300 ns instead of 7100 ns). However, with a dual-processor node the amount of work done by the EU is much smaller, as most of the work can be off-loaded to the SU. Therefore the total EU overhead adds up to only 1500 ns, while in the single-processor case it is 5300 ns. This means that the dual-processor version can achieve a higher throughput.

A dual-processor node design is therefore expected to achieve better overall execution time if the remote access latencies can be masked by multithreading, but could perform worse if this is not the case. In any case, however, the EU costs for a single operation are still quite high compared with the typical costs for loads and stores to the local memory. It is therefore necessary to further reduce this overhead if parallelism is to be exploited efficiently at the level of individual reads and writes.

3.2 Gains through better Hardware Support

On MANNA it turns out that the EU $\leftrightarrow$ SU communication costs dominate the EU overhead. This is in part due to the lack of support for direct cache-to-cache updates, which forces all such communication to go through DRAM. More efficient cache strategies would help, but without additional hardware support it will not be possible to drastically lower the EU overhead.

In order to achieve significant improvements it would be necessary to significantly simplify the communication protocol, e.g. by supporting EU $\leftrightarrow$ SU queues in hardware. Hardware flow control would eliminate the need for test and branch instructions and eliminate some off-chip accesses. Also, bypassing the system bus would reduce the amount of bus collisions. With such hardware support it should be possible to reduce the cost for issuing a request or switching to the next thread to around 100 – 200 ns, less than the typical cost of a cache miss. This is only an estimate, as the exact numbers depends on the amount of hardware spent for supporting communication.

However, some costs are still likely to remain in a dual-processor node configuration. For example, the value returned by a remote load would still have to go through the system bus, as the EU expects the SU to store it in the local memory. Single-processor node designs will not suffer in the same way from inefficiencies due to the memory hierarchy, as no data needs to be transferred from one processor to the other. However, there are also efficiency limits due to the interrupt latency and to the fact that in a single-processor design all the work has to be performed by the main processor. Therefore, the dual-processor approach still seems the most promising for high-performance multiprocessors.

3.3 Benefits of Dual Processors

Even without extensive hardware support, significant gains can sometimes be achieved with a second processor. As an example, we discuss the multithreaded execution of Livermore Loop 7 [9]. The body of this parallel loop is a moderately

complex expression, which reads from several arrays and writes to a different one. For the purposes of our experiment we forced one array to be remote and the others to be local. We also forced the whole loop to execute on a single node, even though it could run in parallel. The only use of parallelism in this case is to mask latencies. In order to experiment with other communication to computation ratios we also artificially increased the amount of communication by fetching the data two, four or more times.

The purpose of this experiment was to measure how well the system is able to hide the communication latencies. In order to compare with the multithreaded code, a version was developed where all the communication is performed at the beginning, before the loop starts. The execution time of that version therefore corresponds to no overlap between communication and computation and no multithreading overhead (because the normal sequential code can be executed after the data has been fetched). In order to get meaningful results we also increased the loop count from 990 to 9900, as with 990 iterations it takes only 190 μs to perform all communication.

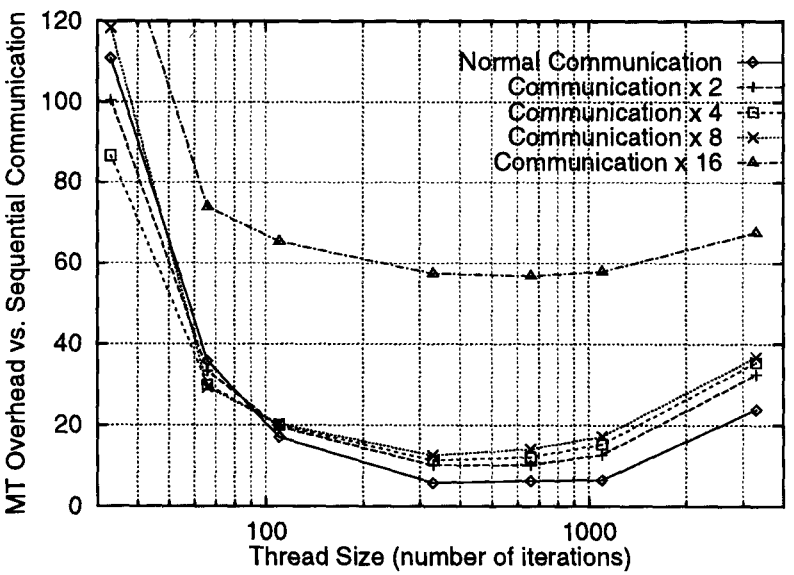

Fig. 3. Percent of communication time masked

Figure 3 compares the total communication overhead for both implementations. A value of 100% means that the total overhead for the multithreaded code was the same as if the data had been fetched all at once at the beginning. A value of 0% would mean that the execution time for the multithreaded code

was the same as for the sequential code (i.e. communication perfectly hidden). This comparison was made for different thread sizes and for different amounts of communication.

With small thread sizes the costs are high, as the communication and multithreading overheads increase. At the other hand of the spectrum, the amount of parallelism becomes too small to efficiently hide the communication. In between, there is a wide range of thread sizes for which a substantial percentage of the communication time could be hidden. The optimum size in this example is 330 iterations, in which case the total communication and multithreading overhead was just 5.8% of the sequential data transfer time. With increased communication the machine behavior remains roughly unchanged until the communication time becomes larger than the computation time (with 16 times the normal amount of data transferred). At that point, of course, it becomes impossible to hide all the communication.

4 Conclusion

We have implemented a multithreading layer on top of the MANNA architecture, a machine based on off-the-shelf RISC processors. Our experiences with the EARTH-MANNA system indicate that it does not introduce substantial overheads. Moreover, the possibility to overlap computation with communication can provide additional performance improvements, significantly reducing the overall costs of remote memory accesses.

The costs associated with multithreading support turned out to be dominated by the EU $\leftrightarrow$ SU communication overhead. In order to efficiently hide latencies at the level of individual loads and stores this overhead has to be drastically reduced. This seems to be possible with appropriate hardware support. On the other hand, the cost of saving and restoring registers during context switches proved to be lower than expected. Multithreading seems to be a promising approach for future multiprocessor systems based on off-the-shelf RISC processors.

5 Acknowledgment

We would like to thank the Natural Sciences and Engineering Research Council (NSERC) for their support of this research. Also, the second author received funding from the Concordia FRDP and the third author received support from FCAR. Special thanks go to the MANNA developers at GMD-FIRST for the MANNA machines and for their technical support. Without them there would be no EARTH-MANNA system. Thanks go also to all the other members of the EARTH group for their contributions to the project.

References

1. Gail Alverson, Bob Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Integrated support for heterogeneous parallelism. In *Multi-*

threaded Computer Architecture: A Summary of the State of the Art, chapter 11, pages 253–283. Kluwer Academic Pub., Norwell, Mass., 1994.

2. Boon Seong Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Lab. for Comp. Sci., Aug. 1994.
3. Gesellschaft für Mathematik und Datenverarbeitung mbH. *MANNA Hardware Reference Manual*. Berlin, Germany, 1993.
4. Herbert H. J. Hum and Guang R. Gao. Supporting a dynamic SPMD model in a multi-threaded architecture. In *Digest of Papers, 38th IEEE Comp. Soc. Intl. Conf., COMPCON Spring '93*, pages 165–174, San Francisco, Calif., Feb. 1993.
5. Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques, PACT '95*, Limassol, Cyprus, Jun. 1995. IFIP WG 10.3, ACM SIGARCH, and IEEE-TCCA. To appear.
6. Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 288–294, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.
7. Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proc. of the 21st Ann. Intl. Symp. on Computer Architecture*, pages 302–313, Chicago, Ill., Apr. 1994.
8. Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992.
9. Frank H. McMahon. The Livermore FORTRAN Kernels: A computer test of numerical performance ranges. Tech. Rep. UCRL-537415, Lawrence Livermore Nat. Lab., Livermore, Calif., Dec. 1986.
10. Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, pages 224–235, San Diego, Calif., May 1993.
11. Peter R. Nuth and William J. Dally. Named state and efficient context switching. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 9, pages 201–212. Kluwer Academic Pub., Norwell, Mass., 1994.
12. Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conf. Proc., 1993 Intl. Conf. on Supercomputing*, pages 251–260, Tokyo, Japan, Jul. 1993.
13. Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.