McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling

Olivier Maquelin *Guang R. Gao*
Herbert H. J. Hum†
Kevin B. Theobald *Xinmin Tian*

ACAPS Technical Memo 99
November 8, 1995

Submitted for publication

†Department of Electrical and Computer Engineering, Concordia University,
1455 de Maisonneuve W., Montréal, Québec H3G 1M8

Abstract

Parallel systems supporting multithreading, or message passing in general, have typically used either polling or interrupts to handle incoming messages. Neither approach is ideal; either may lead to excessive overheads or message-handling latencies, depending on the application. This paper investigates a combined approach, where both are used depending on the circumstances. In the *Polling Watchdog,* a simple hardware extension limits the generation of interrupts to the cases where explicit polling fails to handle the message quickly. As an added benefit, this mechanism also has the potential to simplify the interaction between interrupts and the network accesses performed by the program.

We present a message-handling mechanism designed for the *EARTH-MANNA-S* system, an implementation of the EARTH execution model on the MANNA multiprocessor. In contrast to the original EARTH-MANNA system, this system does not use a dedicated communication processor. Rather, synchronization and communication tasks are performed on the same processor as the regular computations. Therefore, an efficient message-handling mechanism is essential to good performance.

Simulation results, and experiments with a 20-node EARTH-MANNA-S system, show that the Polling Watchdog indeed performs better than either polling or interrupts alone. As measurements with a representative set of benchmarks demonstrate, this mechanism allows the EARTH-MANNA-S system to achieve the same level of performance as the original EARTH-MANNA system.

Contents

List of Figures

1 Introduction

Today, fast microprocessors are commonly used as building blocks in large multiprocessor systems, due to their excellent price/performance ratios. Unfortunately, since these processors are driven by mass markets, they are mostly optimized for use in personal computers and workstations, and offer only limited support for interprocessor communication. Nevertheless, the high development costs of current hardware makes it virtually impossible for custom hardware to compete with "killer micros" [2], especially given the relatively small size of the market for supercomputing. If parallel architects have to "give in" and use off-the-shelf microprocessors, then it is important to study how best to take advantage of their capabilities.

Considerable gains in communication performance can be achieved by making the network interface accessible by user-level code and by using low-latency message passing mechanisms such as Active Messages [17]. As designers try to improve communication performance further, especially for short messages, ways must be found for the system to react to network events quickly and cost-effectively.

We believe that the multithreaded program execution model has great potential for future parallel systems due to its ability to tolerate the communication and synchronization latencies inherent in parallel architectures [1, 4, 6, 12, 15]. As long as there is enough parallelism in an application, a multithreaded architecture can hide these latencies by switching to another thread. However, this necessitates the smooth integration of asynchronous events into the computation [15]. Since multithreaded systems face the same competitive pressures as other multiprocessors, it may be necessary to implement this integration using stock hardware.

In this paper, we present a message-handling mechanism designed for the *EARTH-MANNA-S* system, an implementation of the EARTH execution model [7, 8, 9] on the MANNA multiprocessor [3]. This system, which is described in more detail in section 2.1, implements a multithreading environment on a machine based on Intel i860 XP RISC processors. In contrast to the original EARTH-MANNA implementation [7, 8] which served as an emulation platform for our investigations of EARTH concepts, it does not use a dedicated communication processor. Instead, a single CPU is in charge of both computations and communications. Thus, the experiences of our EARTH-MANNA-S system are generally applicable to multithreaded runtime systems implemented on conventional multiprocessors consisting of single-CPU nodes.

On most machines in which a single CPU performs both functions, one of two mechanisms is used to handle incoming messages. Either there are explicit polls in the application code (added by the user or compiler), or incoming messages generate interrupts in the CPU. However, using either polling or interrupts exclusively can be inefficient, especially when message traffic is high and its irregular pattern is not statically analyzable. If polling is used, but the message frequency is much lower than the polling frequency, the polling overhead will be incurred many times for each message. If, on the other hand, interrupts are used and the message frequency is very high, the cumulative costs of servicing an interrupt for each message can become prohibitive. There is a definite need for a better solution, as the frequency of network events can vary dramatically from program to program and even during program execution.

The approach described in this paper seeks to combine the advantages of both polling and interrupts. A simple extension to the hardware, called a *Polling Watchdog*, generates interrupts only when explicit polling fails to handle incoming messages in a timely manner. In a conventional OS, a *watchdog timer* may be used by the kernel to regulate interrupt generation in order to prevent a process from running too long [16]. Similar mechanisms have also been used in real-time systems to improve response time. The Polling Watchdog proposed in this paper is targeted toward general-purpose multiprocessor architectures with mulithreaded program execution models. It is to be used in conjuction with normal user-level polling to improve the performance of message handling (data communication and synchronization) between multiple threads within one user application, and no OS intervention is required.

While the hardware for the Polling Watchdog has not been implemented, we have modeled its behavior in two ways. In the first method, the MANNA machine on which EARTH-MANNA-S runs has two CPUs per node, thus the other CPU is used to emulate the Polling Watchdog's function. Since this emulated Watchdog may interfere with CPU performance more than would a real Watchdog, we have simulated the proposed hardware extensions using SEMi (*Simulator for EARTH, MANNA and i860*) in our second method. This simulator provides detailed cycle-by-cycle simulation of the MANNA machine. The validity of the simulation results was ascertained by comparing them to the actual machine behavior where possible.

In this paper, we make the following contributions:

- We propose a combined approach for efficient message handling. The *Polling Watchdog* uses interrupts, but limits them to the cases where explicit polling fails to handle messages quickly.
- In the presence of interrupts, some kind of locking mechanism is needed to guarantee atomic access to shared resources. We propose a simple solution for that problem under our scheme.
- We compare experimentally the per-message overheads and latencies of several message-handling mechanisms. Our experiments cover both the simulation of a hardware implementation of the Polling Watchdog and the current implementation in the EARTH-MANNA-S system. Among our findings:
  - In terms of latency, the Polling Watchdog clearly achieves better performance than a naive polling-only approach, and the same performance as an optimal polling-only solution.
  - The overhead per message of our scheme is considerably lower than for an interrupt-based implementation, especially in the presence of fine-grained computations. Polling-only approaches can give even better performance, but only at

the cost of programmer intervention and with a higher sensitivity to variations of the message-arrival rate.

- We also demonstrate the performance gains achieved for a representative set of benchmark programs. These results are very encouraging. Although the EARTH-MANNA-S system does not have a dedicated communication processor, it is able to achieve the same level of performance as the original EARTH-MANNA system.

- We give some preliminary guidelines on how to choose a timeout value for the Polling Watchdog. The effect of the timeout value on execution time is shown for the same set of benchmark programs.

The rest of the paper is organized as follows: the next section briefly describes the EARTH model and the EARTH-MANNA-S system, in particular the challenges facing designers of such a system. Section 3 describes the Polling Watchdog mechanism, as well as the implications for guaranteeing atomicity. Section 4 discusses the performance of several message handling mechanisms, based both on experiments with the EARTH-MANNA-S system and on simulation results. Then, section 5 shows the performance gains achieved for a representative set of benchmark programs and compares these results with the performance of the original EARTH-MANNA system. Finally, related work is mentioned and some conclusions are drawn.

2 The EARTH-MANNA-S System

The issues discussed in this paper were encountered while mapping the EARTH model to a machine with only a single CPU per node. Both the original EARTH-MANNA system and the EARTH-MANNA-S system run on the MANNA multiprocessor, which has two CPUs per node. However, while the EARTH-MANNA implementation dedicates one of them to communication and synchronization, the EARTH-MANNA-S system uses only a single CPU. This second implementation was made in order to assess the usefulness of a dedicated communication processor and to study the problems involved in porting the EARTH model to single-CPU node architectures.

2.1 The EARTH Programming Model

EARTH (*Efficient Architecture for Running THreads*) [8] supports a multithreaded program execution model in which code is divided into threads that are scheduled atomically using dataflow-like synchronization operations [7, 8, 9]. These "EARTH operations" comprise a rich set of primitives, including remote loads and stores, synchronization operations, block data transfers, remote function calls and dynamic load balancing. EARTH operations are initiated by the threads themselves. Once a thread is started, it runs to completion, and instructions

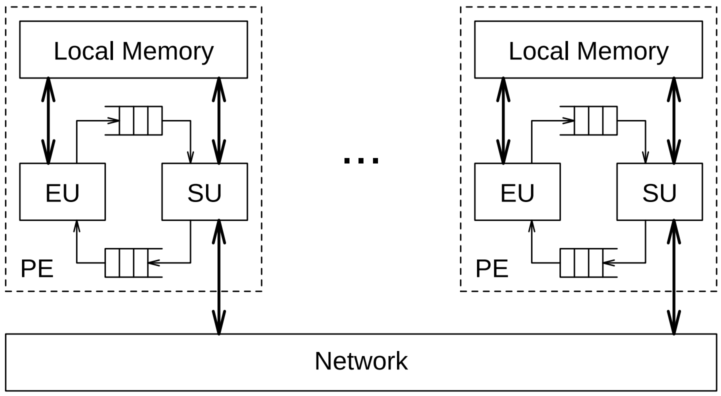

Figure 1: The EARTH Architecture

within it are executed in sequential order.[1] Therefore, a conventional processor can execute a thread efficiently, even when the thread is purely sequential. For this reason, it is possible to obtain single-node performance close to that of a purely sequential implementation, as shown in our earlier work and recapitulated in section 5.

Conceptually, each EARTH node consists of an *Execution Unit* (EU), which executes the threads and a *Synchronization Unit* (SU), which performs the EARTH operations requested by the threads (see figure 1). The current hardware designs for EARTH use an off-the-shelf high-end RISC processor for the EU and custom hardware for the SU. However, other implementations are also possible.

The EARTH model has been implemented on the MANNA 2.0 [3] architecture. Each MANNA node contains two Intel i860 XP RISC CPUs, 32 MB of local memory and a 50 MB/s bidirectional network interface. The interconnection network is based on 16×16 crossbars, which support the full 50 MB/s link bandwidth. Because two CPUs are available per node, one of them is used as the EU, executing code compiled from the user program, while the other CPU runs a fixed set of routines that emulate the functions of the SU. The resulting system is called EARTH-MANNA [8, 11].

2.2 Mapping EARTH to a Single CPU

Since most multiprocessor systems only have a single CPU per node, an obvious question was whether the EARTH model can be implemented with competitive performance and efficiency on such machines, and what key issues and challenges must be addressed. To answer these

[1] Instructions may be executed out of order, as on a superscalar machine, as long as the semantics of the sequential ordering are obeyed.

questions, we did another implementation of the EARTH model on MANNA, called EARTH-MANNA-S, which uses only one of the two CPUs to perform the functions of both the EU and SU.

With only a single CPU to execute both the program code and the multithreading support code, it is necessary to find an efficient way to switch from one to the other. For initiating external requests and even for context switching this is not too much of a problem, as this occurs at specific locations in the code. The operations can therefore be expanded inline or implemented as function calls. Efficient user-level access to the network can dramatically reduce the overheads for message sends [17] and the costs of context switching can also be kept quite low with appropriate compiler support [11]. However, asynchronous reception of messages is more difficult, as it has to be integrated into the flow of control dynamically, at run-time. This problem is not limited to the EARTH model. Others have also encountered similar difficulties [15].

Two main criteria determine the efficiency of a message handling mechanism: the overhead per message and the average handling latency (time to respond to an incoming message). Low per-message overheads are crucial when large numbers of short messages are exchanged. This can easily be the case with the EARTH-MANNA-S system, as the efficient support for multithreading makes it possible to exploit parallelism at a fine level of granularity, down to thread lengths of just a few hundreds of instructions. Low response times are important when there is little parallelism. Multithreading systems are able to tolerate long response times by running other threads while the communication is in progress, but only if enough threads are available to hide all the delays.

The questions mentioned above can therefore be narrowed down to the following: *Is it possible to keep both the per-message overhead and the latency of message handling to sufficiently low values on machines with a single off-the-shelf microprocessor per node*? Low latencies can be achieved with interrupts, but on today's RISC CPUs this mechanism is quite expensive. The lowest overheads are achieved with explicit polling, but how do we then guarantee short latencies without compromising sequential execution performance? The rest of this paper will address that question by describing the message handling mechanism implemented in the EARTH-MANNA-S system.

3 The Polling Watchdog

Explicit polling and interrupts both have their strengths and weaknesses. Polling has been used for many message-passing implementations designed to support small messages efficiently, such as the Active Messages implementation on the CM-5 [15]. However, interrupts have the distinct advantage of being generated only when a message arrives. Poll instructions are always executed, whether or not a message is there to be processed. Therefore, polling becomes wasteful when the frequency of messages is much lower than the polling frequency. Addressing this

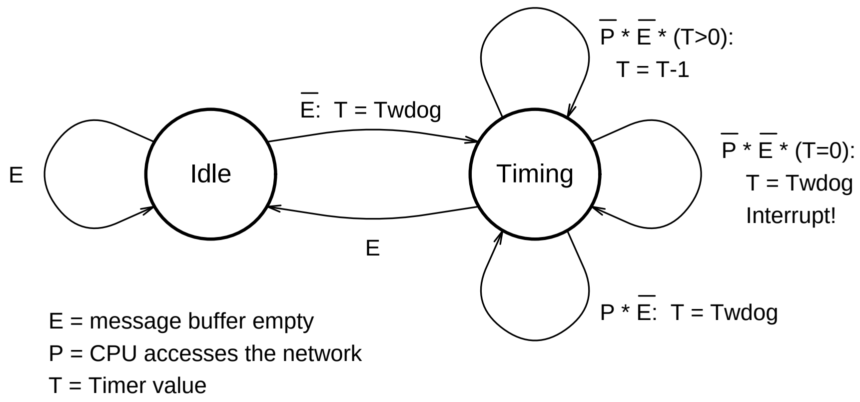

Figure 2: State Diagram of the Polling Watchdog

problem simply by reducing the polling frequency does not always lead to better performance, because even when the number of messages is low, it might be necessary to respond to these messages quickly in order to achieve good overall efficiency.

This may not be a problem for systems such as TAM implementation on the CM-5 [5] (we call this *TAM/CM-5*), where polling once per thread is sufficient due to the fine granularity of that model. In that particular case, polling can be integrated into the other multithreading support operations. However, for other models such as EARTH, where larger threads are common, polling once per thread is not always sufficient. Additional polling instructions could be added at appropriate locations in the code, either by the programmer or by the compiler. However, this can be difficult to implement in practice. If at all possible, such machine-specific issues should be hidden from the user.

3.1 Mixing Polling and Interrupts

If polling and interrupts both have advantages and disadvantages, would it be possible to let the machine switch from one mode to the other at run-time, depending on circumstances? To answer this question, we implemented the *Polling Watchdog* mechanism. The EARTH-MANNA-S system polls the network during context switches, as is done, for instance, in TAM/CM-5. When a message arrives, a timer starts counting. If the message is not removed from the network through polling within the watchdog timeout T_{wdog}, the watchdog interrupts the CPU.[2] This is shown in the state diagram in figure 2, which performs transitions every clock cycle (T_{wdog} is measured in clock cycles). Figure 3 illustrates this graphically. In this example, a message arrives at time *t1* and is handled through polling at time *t2*. At time *t3* a second message arrives, which is handled through an interrupt at time *t4* because the network was not polled quickly enough. Note that the watchdog timer is reset, but continues to count until the

[2] A similar mechanism can be seen in conventional operating systems, in which a *watchdog timer* is used to prevent a process from running too long [16].

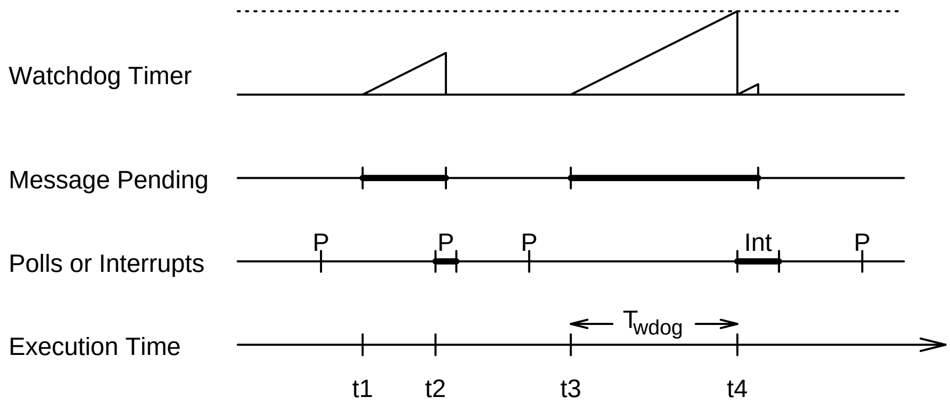

Figure 3: Mixing Polling and Interrupts

message is actually removed from the network. If the interrupt code fails to empty the network buffers, another interrupt will be generated. Section 3.2 explains the reason for this behavior.

In spite of its simplicity, such a mechanism is not available on current multiprocessors. The EARTH-MANNA-S implementation solves this shortcoming by using the second CPU of each MANNA node, which would otherwise sit idle, to emulate the watchdog timer. (One would normally not want to dedicate a whole CPU for such a trivial task.) Furthermore, this emulation is not perfect, as the second CPU needs to use the memory bus, which is shared by the two CPUs, whenever it checks the status of the network receiver. This extraneous bus usage can degrade the performance of the main CPU.

The watchdog timeout value T_{wdog} is an important run-time parameter. A bad choice for that value can eliminate most of the advantages of using a Polling Watchdog. If T_{wdog} is too low, too many interrupts will be generated, increasing the message handling overheads. If, on the other hand, T_{wdog} is too large, the resulting message handling latencies may become prohibitive. In general, the optimal value for T_{wdog} will depend on the program that is being executed. It is, however, possible to derive a lower limit that is useful in most cases.

Because the Polling Watchdog delays the generation of interrupts by T_{wdog}, the maximum frequency of network interrupts will be $1/T_{wdog}$. By increasing the value of T_{wdog} it is therefore possible to reduce the frequency of interrupts, and therefore their impact on overall execution time. If C_{int} denotes the cost of an interrupt, the interrupt overheads will account for a fraction of at most C_{int}/T_{wdog} of the execution time.

In the EARTH-MANNA-S system, C_{int} is $4.5\,\mu$s and the default T_{wdog} is $50\,\mu$s. Interrupt dispatch costs can therefore amount to at most 9% of total execution time. However, the typical impact of interrupts on execution time is normally much lower. This worst case can only occur when a node that executes some long thread receives exactly one message during every T_{wdog} interval. While periodic message arrivals are quite possible, other arrival patterns are also to be expected, such as random arrivals or bursty traffic. Bursts of messages can especially benefit from the Polling Watchdog, as a single interrupt will be able to handle several messagess with one interrupt, thus reducing the per-packet overhead.

Setting a limit on the interrupt overhead defines a lower limit for T_{wdog}, but what about the upper limit? That limit depends on the amount of independent computations that can be performed after a message has been sent. If enough parallelism is available, a multithreaded system can hide substantial delays without noticeable performance degradation. On the other hand, if no parallelism is available it might be necessary to set T_{wdog} to a very low value to get optimal performance. The optimal timeout value will in general depend on the program. However, our experiments seem to indicate that there is in fact a range of values within which system performance is quite robust (see section 5).

3.2 Atomicity Requirements

Mixing polling and interrupts has some important consequences for the run-time system, as the code that does the polling could be interrupted. Therefore, some measures have to be taken, as it is necessary to guarantee atomic access to shared resources, such as the run-time system queues and the network interface. In a conventional implementation this would typically be handled by disabling interrupts while critical parts of the run-time system are active. This is the case in the TAM implemenation on the J-Machine (TAM/J-Machine) [14].

The drawback of this approach is that interrupts have to be disabled and re-enabled each time the network is polled, a message is sent or some other access to the critical data structures is performed. If threads are short, this can add up to a significant portion of execution time. For example with TAM/J-Machine, enabling and disabling interrupts accounts, on average, for 4% of the execution time [15].

However, with the Polling Watchdog a different approach is also possible. Instead of letting the run-time system disable interrupts, the interrupt handler can check if the run-time system is in a critical section and simply return without doing anything if this is the case. With a conventional interrupt mechanism this is not possible, as the trap handler has to first clear the source of the interrupt, i.e. handle the message. However, with the Polling Watchdog another interrupt will occur after a T_{wdog} interval. Also, the run-time system is likely to poll the network, and therefore handle the message, whenever it executes such code.

This approach was implemented in an efficient manner in the EARTH-MANNA-S system. Because the parts of the run-time system that deal with the network are self-contained, it is sufficient for the interrupt handler to test the address where the interrupt occurred. If the program counter lies within a specific address range, the interrupt handler assumes that a critical section was interrupted and discards the interrupt. Therefore, no code had to be added to signal the beginning and end of critical sections.

3.3 EARTH-MANNA-S Implementation

The EARTH-MANNA-S implementation does not implement the watchdog mechanism exactly as described in section 3.1. In order to do that it would have to closely monitor the network state and measure how long messages have been waiting. However, the EARTH-MANNA-S watchdog is implemented in software on the second CPU of each MANNA node. It accesses the network state through the system bus, which is shared between the two CPUs. Therefore, accessing the network state too frequently could cause bus conflicts and degrade performance.

For that reason, the EARTH-MANNA-S implementation limits the frequency of such accesses to only one per timeout interval. If at that point the receive buffer is nonempty, an interrupt is generated. Interrupts will therefore occur at multiples of the timeout interval. Also, interrupts will be generated even if the message in the receive buffer has just arrived.

The behavior for individual messages can thus be quite different from the proposed model. However, even though the waiting time is variable, it is still bounded. Messages will wait for at most T_{wdog} before an interrupt occurs. The main difference is that this modified mechanism is more likely to generate interrupts unnecessarily. Fortunately, it turns out not to be a serious drawback for most applications, as the results in section 5 demonstrate. For a more detailed comparison, section 4 shows both the performance of the EARTH-MANNA-S watchdog and simulation results showing the behavior of a true Polling Watchdog.

4 Message Handling Performance

This section and the next present our experimental results. In this section, we examine specific measurements relating to message handling, e.g., latency and overhead, and compare the message-handling mechanisms previously discussed. Empirical measurements are made under a synthetic workload. In the next section, we measure the overall performance improvements produced by the Watchdog for real benchmarks.

This section shows the per-message overhead and average latency achieved with 6 different message handling mechanisms, including the Polling Watchdog. In order to show the effect of thread length on these numbers, our test program generates threads of random size, but with a preset average length. While these threads are executed on node 1, node 0 uses remote read operations to access data located on node 1. Between these read operations, a random delay, again with known average value, is inserted between to control the message frequency. The time to handle the read requests, as well as the overheads incurred on node 1 were measured for the following system configurations:

- *Default polling only*: The network is polled only during context switches. This is the most basic configuration, similar to TAM/CM-5.

- *Polling added in threads*: For these measurements, polling code is added to the threads that simulate the workload. This code ensures that the network is polled at least once every 50 μs.

- *Interrupt driven*: This version uses interrupts to handle all external messages. The trap handling cost has to be paid for each incoming message.

- *EARTH-MANNA-S watchdog*: These results use the default configuration of the EARTH-MANNA-S system. The second CPU polls the network every 50 μs, which leads to an average delay of 25 μs (for evenly distributed message arrivals) until an interrupt is generated.

- *Polling Watchdog*: This is for a hardware implementation of the Polling Watchdog, as described in section 3.1. These results were gained with SEMi, the EARTH-MANNA simulator. The watchdog timeout was set to 25 μs.

- *EARTH-MANNA*: This is the performance of the original EARTH-MANNA system, which dedicates one processor for communication. It is thus not necessary for the main CPU to poll the network or to be interrupted.

4.1 Message Latency and Overhead

Figure 4 shows the average latency of a remote memory access for the different message handling mechanisms. As expected, the results for both the interrupt-driven and the original EARTH-MANNA implementation (12 μs and 7 μs respectively) are independent of the thread length. Also as expected, the performance of default polling degrades rapidly when the average thread size increases. Because in this case no polling is performed during the threads, the latency is proportional to the thread length. On the other hand, the remaining three mechanisms all limit latencies to about 30 μs. For an average thread length of less than 10 μs these three mechanisms even achieve lower latencies than an interrupt-based system. Polling added performs slightly better for long threads, as the lower polling overhead also leads to shorter latencies.

Figure 5 shows the per-message overhead incurred for these experiments. The average interval between read requests is 50 μs. As expected, using only the default polling leads to the lowest costs: approximately 1 μs per message. The interrupt-driven version has the highest overhead, mostly due to the 4.5 μs needed to dispatch an interrupt. For EARTH-MANNA the costs are practically nonexistent; they are due only to bus access conflicts, as incoming messages are handled by the communication processor.

The overhead of the Polling Watchdog and the EARTH-MANNA-S watchdog depend on the average thread length. For average lengths up to 10 μs most messages are handled through polling. The resulting overhead is therefore close to that of default polling. The EARTH-MANNA-S implementation has a larger overhead in this range, as some interrupts are

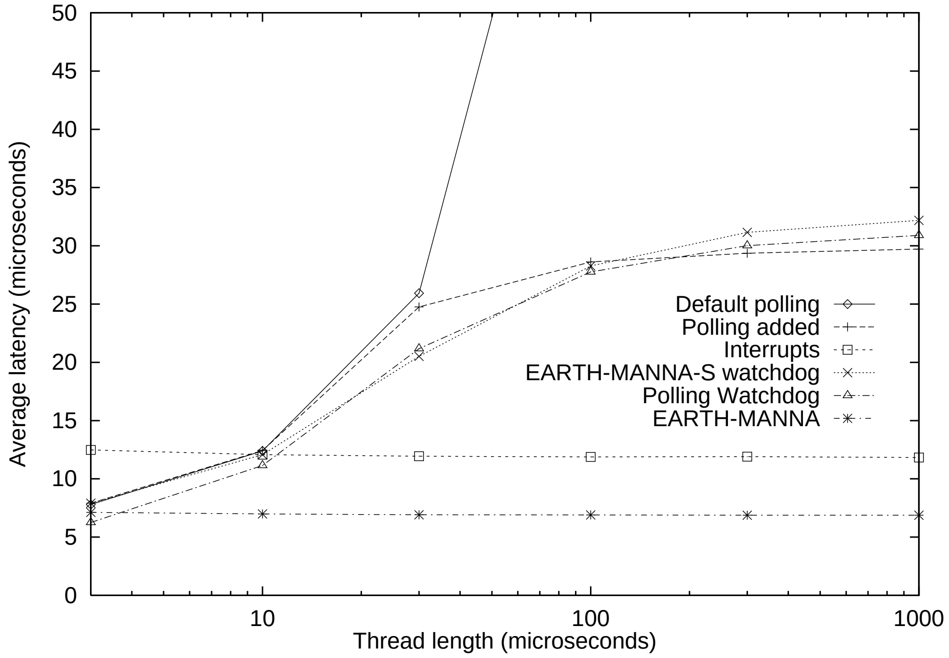

Figure 4: Average Latency for a Remote Read

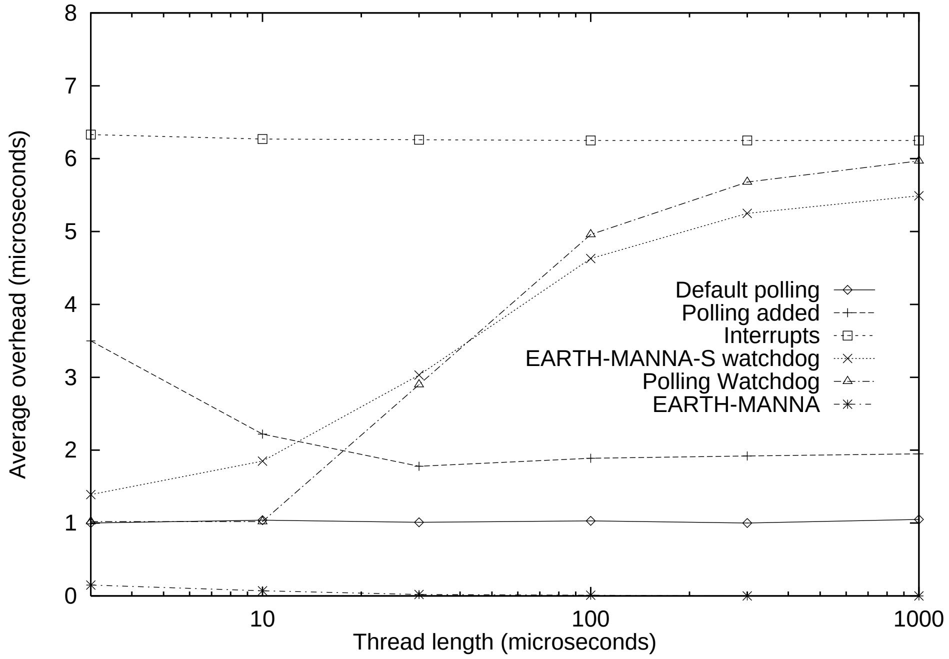

Figure 5: Average Overhead Due to Remote Read

generated unnecessarily. For larger threads, both show approximately the same performance, which is close to that of an interrupt-based system as most messages are handled through interrupts. The slightly higher values for the Polling Watchdog are due to inaccuracies of the simulation.[3]

As the results in figure 4 and figure 5 show, adding code to poll the network at regular intervals can give very good overall results. Two important observations should be made, however. Firstly, it was necessary to modify the code (by hand) to add these polling instructions. In other words, the user had to deal with machine-specific details in order to improve performance. Secondly, in these measurements the polling frequency matches exactly the message arrival frequency. This can be difficult to achieve in practice. Section 4.2 shows how the overheads change when this is not the case.

4.2 Impact of Message Arrival Frequency

The overhead of dispatching an interrupt is basically constant. This cost can be amortized to some extent if several messages are handled with the same interrupt. The probability of that occurring is especially high with the Polling Watchdog, as the timeout period provides substantial time for other messages to arrive.

The cost of a polling operation mainly consists of three components:

- cost of keeping track of when to poll
- cost of polling the network
- cost of handling the message, if any

The cost of handling the message can of course not be avoided. On the EARTH-MANNA-S system it typically takes about a microsecond to receive a short message. The cost of polling the network is about 400 ns, mainly due to the time needed to read the network state. Keeping track of when to poll encompasses all additional costs resulting from the addition of polling code to the original program. In our experiments, most of the execution time of a thread is spent in a tight delay loop. This loop had to be split into two nested loops in such a way that the inner loop takes about $50\,\mu$s to execute. This caused some overheads, which proved to be more important for smaller threads, as can be seen from figure 5.

In practice it will be difficult to insert polling code in exactly the right locations or to know exactly with which frequency the network should be polled. This would be even more difficult

[3] We are currently investigating the reason for these differences and hope to provide more accurate results in a future version of this paper.

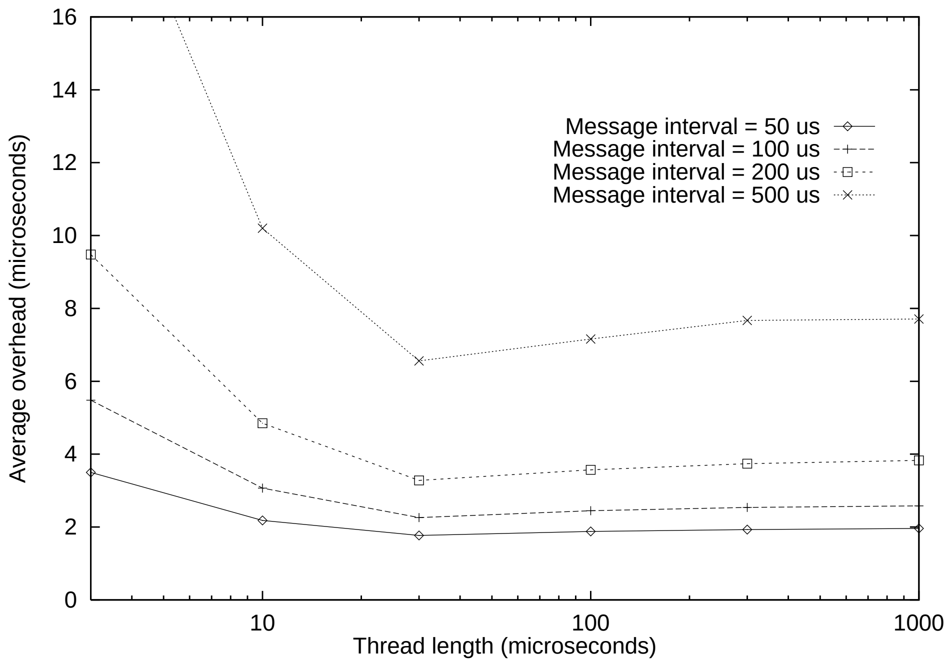

Figure 6: Average Overhead for Added Polling

for a compiler to determine if it was made responsible for inserting polling code automatically. Therefore, conservative assumptions will have to be made to make sure that the network is polled frequently enough. Thus, the polling frequency will normally not be the same as the message arrival frequency.

Figure 6 shows how the performance for added polling changes as the message arrival frequency decreases. This experiment uses the same code as section 4.1 for the simulated workload. The network is again polled once every $50\,\mu$s, but the interval between messages is varied from $50\,\mu s$ to $500\,\mu s$. As the number of poll operations per message increases (with increasing message intervals), the per-message overhead also increases. In contrast, figure 7 shows that the per-message overhead for the Polling Watchdog remains constant, independently of the message arrival frequency.

The polling overhead can become arbitrarily large as the difference between polling frequency and message arrival frequency increase. In our experiments the Polling Watchdog performs better than polling as soon as more than 6 polls are necessary to handle a message. The point at which polling becomes inefficient depends on the difference between the cost of handling an interrupt (C_{int}) and the cost of polling (C_{poll}). Polling is always less efficient than interrupts when more than C_{int}/C_{poll} polls are performed per message.

On the EARTH-MANNA-S system it takes approximately $4.5\,\mu$s to service an interrupt and 400 ns to poll the network. Therefore, C_{int}/C_{poll} is approximately 11. The per-message

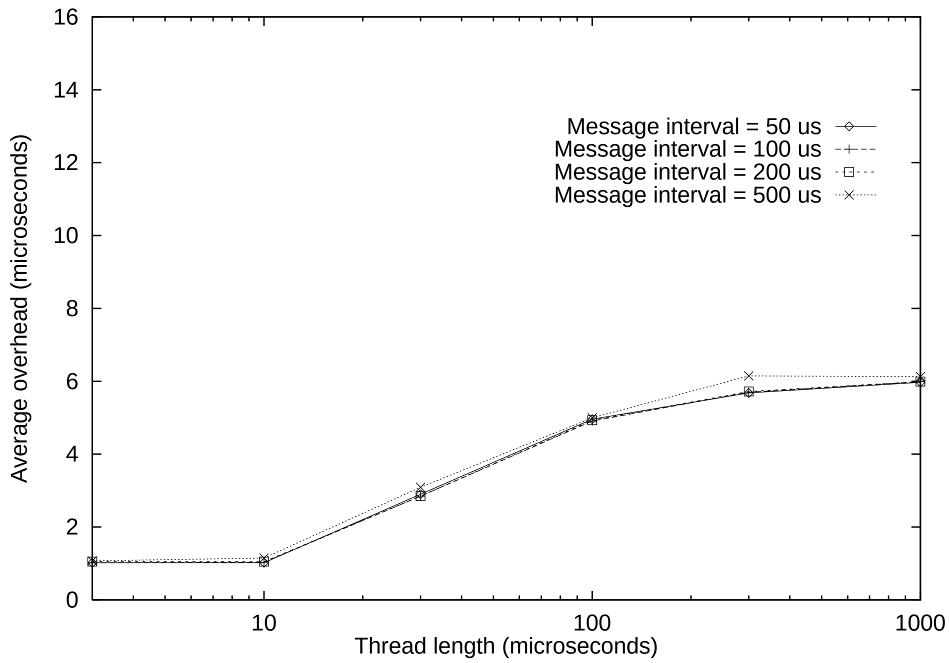

Figure 7: Average Overhead for Polling Watchdog

overhead for added polling will thus only be better if at most 11 polls are made per message. In many cases the upper limit will be even lower (e.g. 6 in our experiments). This can be difficult to achieve in practice, as the message arrival frequency can vary dramatically during program execution. Therefore, even though the Polling Watchdog might not always give the best performance, it will normally be much easier to use such a mechanism than to add polling instructions to the code.

5 Benchmark Results

A set of seven representative benchmarks was used to show the effect of the Polling Watchdog mechanism on program performance. These benchmarks vary widely in size, execution time, communication patterns and load distribution strategies. Some use static load and data distribution while others take advantage of dynamic load balancing. The average thread length varies from tens of microseconds to hundreds of milliseconds.

TOMCATV is a floating-point SPEC89/92 benchmark that performs data-parallel computation on a pair of 257×257 meshes. *Matrix Multiply* multiplies two pre-distributed 480×480 matrices. *2D-SLT* uses a semi-Lagrangian time discretization scheme to simulate a global atmospheric model; our experiments used 80×80 meshes. The *POV-Ray* raytracer renders a complex scene as a 256×256 pixel image. The *Protein Folding* program finds all possible polymers of a $3 \times 3 \times 3$ cube. The *TSP* benchmark computes the optimal route for a traveling salesman through 10 cities. Finally, *Paraffins* enumerates all distinct isomers of paraffins of size up to 23.[4]

Figures 8 and 9 show the performance of the EARTH-MANNA-S system, while figure 10 shows the performance of the original EARTH-MANNA system. The results for added polling were not measured, as this would have necessitated significant changes for some of the benchmarks. These graphs show the machine utilization, i.e. the achieved speedup divided by the number of nodes. These numbers are measured relative to the execution time of the original sequential code, not to the single-node performance of the multithreaded code. In other words, a value of 0.9 for 20 nodes means that the multithreaded code on a 20-node machine runs 18 times faster than a purely sequential implementation.

These benchmarks were developed with the dual-processor version of the EARTH-MANNA system, which dedicates one of the CPUs to communication and synchronization tasks. Therefore, some of them achieve poor performance when running in an environment with default polling only (see figure 8). This is especially true for POV-Ray, Protein Folding and Paraffins, which rely heavily on dynamic load balancing to achieve good speedups. Load balancing is poor when the watchdog is disabled, as the long message handling latencies slow down the rate at which load information and work units can be exchanged. The performance of TOMCATV also suffers, not because of load balancing but because block data transfers stall the processors. Because buffer space in the MANNA network is quite limited, the sender of a large block of data has to wait for the receiver to start receiving before it can terminate the send operation. The other benchmarks suffer less, or not at all from the lack of interrupt support. Notably, Matrix Multiply and 2D-SLT are virtually unaffected by the lack of a watchdog (see figure 9).

All these results were obtained with the default watchdog timeout of $50\,\mu$s. The good performance achieved with these programs shows that a multithreading system does not necessarily cause excessive overheads, even when conventional hardware is being used. These results also show that the lack of a dedicated communication processor does not have to be a significant drawback if a mechanism like the Polling Watchdog is used[5].

The influence of the EARTH-MANNA-S watchdog on execution time can be seen more clearly in figure 11. This figure shows the speedup (relative to sequential code) for 20 nodes as a function of the watchdog timeout. The range of timeout values for this experiment spans five orders of magnitude, from $1\,\mu$s, which is shorter than the time to handle an interrupt, all the way up to 100 ms.

As expected, the speedup of most benchmarks degrades with long timeout values. However, there are also exceptions such as 2D-SLT. In this case, the default polling turns out to be

[4]The implementations of most of these applications are described in greater detail in our earlier work [7].

[5]More detailed studies are needed to quantify exactly the usefulness of a dedicated communication processor, as some of the overheads incurred by this implementation are specific to MANNA. This will be the scope of a further paper.

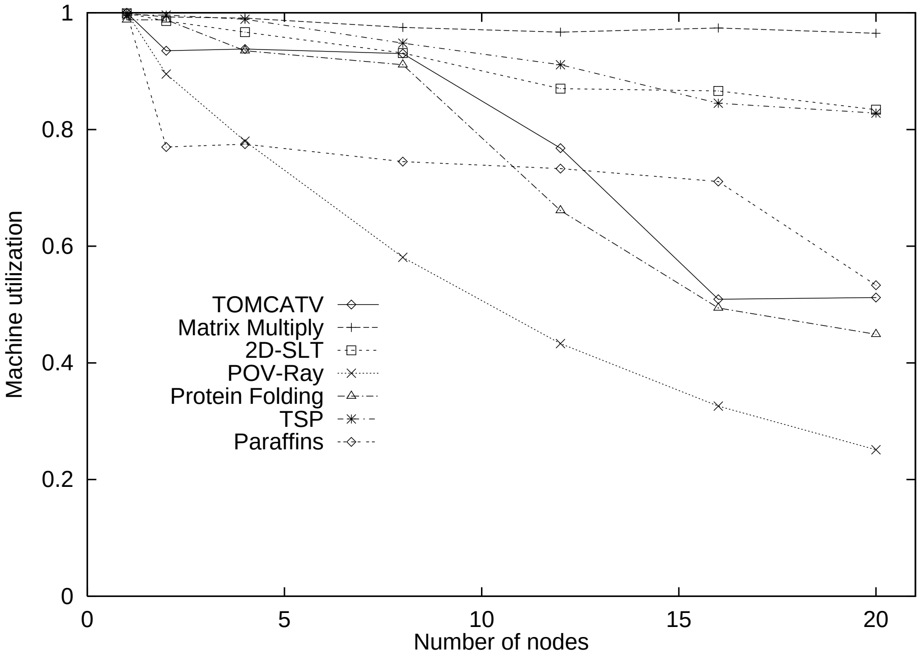

Figure 8: Machine Utilization with Watchdog Disabled

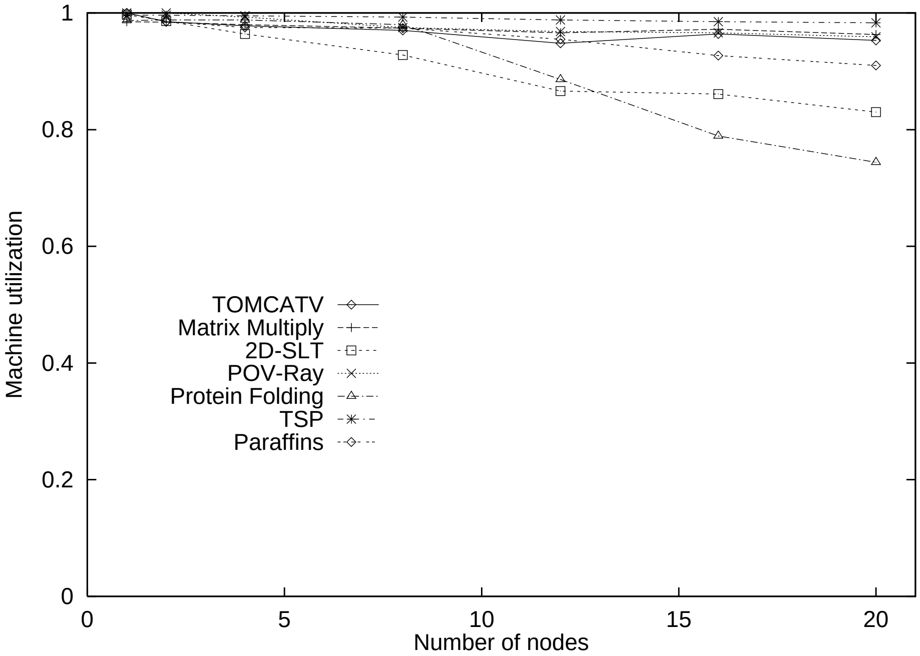

Figure 9: Machine Utilization with Watchdog Enabled

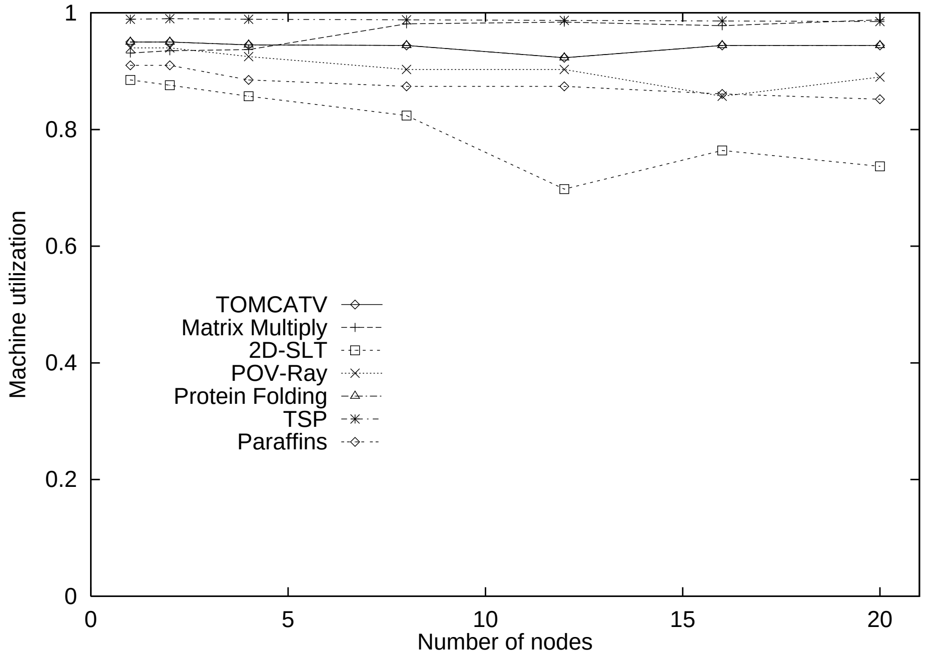

Figure 10: Machine Utilization for Original EARTH-MANNA

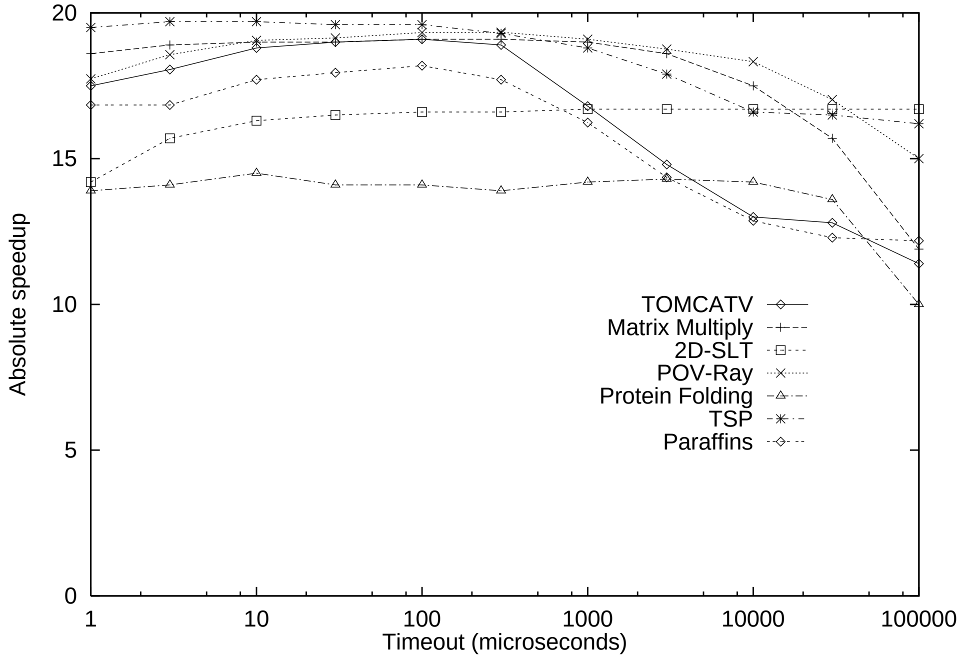

Figure 11: 20-Node Speedup as Function of Watchdog Timeout

perfectly sufficient to handle incoming messages. Due to the way the algorithm is implemented, receiving nodes are usually idle when messages arrive. Because idle nodes constantly poll the network, most messages are handled immediately.

Matrix Multiply also shows a very interesting behavior. From the results in figure 11 it would seem that disabling the watchdog would lead to very bad performance. However, this is not the case, as the excellent speedups in figure 8 demonstrate. The reason for that strange behavior is again due to the structure of the algorithm. Each node executes a loop that initiates a block data transfer at the beginning of the loop body. When only polling is used, all data transfers take place simultaneously at the end of the loop body. On the other hand, when the watchdog is enabled the execution of the body will be interrupted so the data transfer can be performed. Unfortunately, if the watchdog timeout is very large the sender will have to wait for a long time until the other party responds.

For timeout values that are less than $30\,\mu$s the performance of most benchmarks degrades. This is due to the larger number of interrupts that have to be handled, and also to inefficiencies that are specific to the EARTH-MANNA-S implementation. Because that system does not exactly implement the watchdog mechanism, as was explained in section 3.3, interrupts can be generated unnecessarily. Also, it is necessary for the second CPU to access the bus, which can have a significant effect on program performance.

It is encouraging to see that most benchmarks show a very flat optimum and that there is a large range of values within which the system performance is quite robust. For this set of benchmarks good results are achieved with timeouts ranging from $10\,\mu$s to $300\,\mu$s. Lower timeout values will tend to generate an excessive amount of interrupts. Higher values, on the other hand, will increase the average latencies and therefore reduce the efficiency of the load balancer or cause stalls during data transfers.

It should be remembered that using the Polling Watchdog mechanism does not always improve performance. A performance gain compared with an implementation that uses polling is only possible if there are long threads which would cause excessive delays. On the other hand, a performance gain compared with an implementation that uses interrupts is only possible if the message arrival frequency is high. However, based on our experiments we believe that this mechanism can provide a single solution that works well for most programs, independently of their communication behavior.

6 Related Work

The projects most relevant to our paper are TAM/CM-5 [15, 17], the J-Machine [12], Alewife [1], Start-NG [4], EM-X [10], and RWC-1 [13]. We believe that the asynchronous message reception mechanism proposed in this paper can also be applied to non-multithreaded architectures. To the best of our knowledge, this paper is the first to propose the use of a *Polling Watchdog* to handle incoming messages efficiently.

In TAM/CM-5 [15, 17], message arrival is detected through polling. Message reception using interrupts is also possible, but this was deemed impractical due to the high interrupt costs, about 10 times higher than polling [15]. There does not seem to be an urgent need for more efficient interrupt support, as the typical thread length, and therefore the polling interval, is quite short. However, other software using Active Messages on the CM-5 might benefit from such support.

The MIT J-Machine supports two-level scheduling in hardware. However, because interrupts are kept disabled for a significant portion of thread execution, the J-Machine can almost be thought of as using polling [15]. This approach makes it easier to guarantee the atomicity of accesses to the system queues, as the compiler can control when the computation can be interrupted. However, disabling interrupts for long periods of time could increase message latencies. This machine will not benefit from the Polling Watchdog mechanism, as its fast hardware dispatch is more efficient than polling.

The MIT Alewife machine provides a fast message-passing interface which works with the special hardware support in the Sparcle processor. Message reception can use either polling or interrupts. Using interrupts, the time to enter a system-level handler is approximately 35 cycles. This is significantly less than the corresponding numbers for the CM-5, but still more expensive than using polling. This machine might benefit from the Polling Watchdog mechanism, but not as much as systems based on off-the-shelf microprocessors.

In the StarT-NG multicomputer developed at MIT, the PowerPC 620 is expected to poll the NIU to see if messages have arrived. The NIU can interrupt the 620 processor to avoid the polling overhead if it is known that messages arrive very infrequently [4]. As of our knowledge, however, no support for a combination of polling and interrupts is planned.

The EM-X [10] supports interprocessor communication with small and simple packets. A special hardware unit (MU) is employed to handle messages and prepare for thread invocation; thus the execution unit (EXU) can be dedicated to computation tasks.

The RWC-1 parallel machine uses the RICA (*Reduced Interprocessor-Communication Architecture*) architecture [13] based on the Continuation Driven Execution Model. RICA has a hardware Activator Queue. The Activator Queue on RICA can receive and supply activators asynchronously with the interconnection network and Thread Executor. Messages are not received and sent by interrupt or by polling, but by a hard-wired data transfer. The Activator queue can be seen as a network interface of the PE.

7 Conclusions

Today's fast microprocessors are attractive building blocks for the next generation of multiprocessor systems. However, current high-end microprocessors are mainly optimized for sequential performance and lack the ability to integrate external events into the flow of computation efficiently. Interrupts are typically an order of magnitude more costly than explicit polling.

The proposed Polling Watchdog combines both polling and interrupts into a single unified mechanism. This combined approach leads to lower overheads than a purely interrupt-based implementation, but still guarantees that message latencies are bounded. This is the case even when the message arrival frequency is not predictable, or varies during execution, a situation which is difficult to handle efficiently with polling only. Furthermore, no modifications to the program code are necessary. Only minor modifications to the network interface are necessary to implement such functionality in hardware.

A variant of the Polling Watchdog mechanism was implemented in the EARTH-MANNA-S system, leading to significant performance improvements. The performance of the resulting system is similar to that of the original EARTH-MANNA system, which uses a dedicated communication processor to handle incoming messages. Moreover, there is a range of timeout values for which overall performance seems to be quite robust. These results were achieved in a multithreading environment, but are not necessarily limited to such systems. Other systems that support fine-grained asynchronous message passing might also benefit from a Polling Watchdog.

8 Acknowledgments

We would like to thank the Natural Sciences and Engineering Research Council (NSERC) for their support of this research. Also, the third author received funding from the Concordia FRDP, and the second and third authors received support from FCAR. Special thanks go to the MANNA developers at GMD-FIRST for lending us a 20-node MANNA system. Thanks go also to all the other members of the EARTH group for their contributions to the project.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, pp. 2–13, ACM SIGARCH and IEEE Computer Society, June 22–24, 1995. *Computer Architecture News,* 23(2), May 1995.

[2] Eugene Brooks, "The attack of the killer micros." Presentation in the Teraflop Computing Panel Discussion at Supercomputing '89, Reno, Nevada, November 1989.

[3] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat, "Latency Hiding in Message-Passing Architectures," in *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, pp. 704–709, IEEE Computer Society, April 26–29, 1994.

[4] Derek Chiou, Boon S. Ang, Robert Greiner, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton, "StarT-NG: Delivering Seamless Parallel Computing," in *Proceedings of the First International EURO-PAR Conference* (Seif Haridi, Khayri Ali, and Peter Magnusson, eds.), no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden, pp. 101–116, Springer-Verlag, August 29–31, 1995.

[5] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken, "TAM – A Compiler Controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[6] Jack B. Dennis and Guang R. Gao, "Multithreaded Architectures: Principles, Projects, and Issues," in *Multithreaded Computer Architecture: A Summary of the State of the Art* (Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, eds.), ch. 1, pp. 1–72, Norwell, Massachusetts: Kluwer Academic Publishers, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[7] 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 Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu, "The Multi-Threaded Architecture Multiprocessor," ACAPS Technical Memo 88, School of Computer Science, McGill University, Montréal, Québec, December 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[8] 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 *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95* (Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, eds.), Limassol, Cyprus, pp. 59–68, ACM Press, June 27–29, 1995.

[9] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao, "Building Multithreaded Architectures with Off-the-Shelf Microprocessors," in *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, pp. 288–294, IEEE Computer Society, April 26–29, 1994.

[10] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi, "The EM-X Parallel Computer: Architecture and Basic Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, pp. 14–23, ACM SIGARCH and IEEE Computer Society, June 22–24, 1995. *Computer Architecture News,* 23(2), May 1995.

[11] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao, "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," in *Proceedings of the First International EURO-PAR Conference* (Seif Haridi, Khayri Ali, and Peter Magnusson, eds.), no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden, pp. 117–128, Springer-Verlag, August 29–31, 1995.

[12] Michael D. Noakes, Deborah A. Wallah, and William J. Dally, "The J-Machine Multicomputer: An Architectural Evaluation," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 224–235, ACM SIGARCH and IEEE Computer Society, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.

[13] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato, "Super-Threading: Architectural and Software Mechanisms for Optimizing Parallel Computation," in *Conference Proceedings, 1993 International Conference on Supercomputing*, Tokyo, Japan, pp. 251–260, ACM, July 20–22, 1993.

[14] Ellen Spertus and William J. Dally, "Evaluating the Locality Benefits of Active Messages," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Santa Barbara, California, pp. 189–198, July 19–21, 1995. *SIGPLAN Notices,* 30(8), August 1995.

[15] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 302–313, ACM SIGARCH and IEEE Computer Society, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.

[16] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1987.

[17] Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 256–266, ACM SIGARCH and IEEE Computer Society, May 19–21, 1992. *Computer Architecture News,* 20(2), May 1992.