An Implementation of a Hopfield Network Kernel on EARTH

José N. Amaral, Guang Gao, Xinan Tang

Abstract

EARTH is a multithreaded program execution and architecture model that hides communication and synchronization latencies through fine-grain multithreading. EARTH provides a simple synchronization mechanism: a thread is spawned when a pre-specified number of synchronization signals are received in its synchronization slot – signaling the fact that all dependences required for its execution are satisfied. This simple synchronization mechanism is an essential primitive in Threaded-C – a multithreaded language designed to program applications on EARTH. The EARTH synchronization mechanism has been efficiently implemented on a number of computer platforms, and has played an essential role in the support of a large number of parallel applications on EARTH.

An interesting open question has been: is such a simple mechanism sufficient to satisfy the synchronization needs of the large set of applications that EARTH can implement? Or could the EARTH programming model benefit from the implementation of more elaborate synchronization mechanism? In such case, what are the benefits and tradeoffs of adding this mechanisms to EARTH?

This paper describes the implementation of I-structures under the EARTH execution and architecture model. An I-structure is a data structure that allows for the implementation of a *lenient* computation model. A read operation can be issued to an element of an I-structure before it is known that the corresponding write operation has produced the value. We also introduce a new parallel kernel based on the Hopfield Network and demonstrate how the I-structure support on EARTH can be utilized. We finish presenting a complete Threaded-C program to solve the Hopfield kernel is also presented.

Keywords

Multithreaded Programming, EARTH, I-structures, Hopfield Network.

I. Introduction

The EARTH multithreaded architecture was designed to effectively hide communication and synchronization latency and thus support scalable parallel applications. One of the advantage of the EARTH model is that it can be efficiently implemented using off-the-shelf commercial processors and components [16], [17], [18]. The EARTH architecture and program execution model was first implemented on the MANNA machine [6]. Now, it has been successfully ported to parallel machines such as the IBM SP-2 [7], and a network of affordable computers running the Linux operating system, Beowulf[21], [22]).

Threaded-C is the language used to program the EARTH architecture in the different platforms. Threaded-C implements support for EARTH multithread programming through a set of extensions to the standard C language. Threaded-C offer explicit support to multithreaded operations, such as

Computer Architecture and Parallel Systems Laboratory, University of Delaware, Newark, DE, USA. *http://www.capsl.udel.edu*, emails: *amaral@capsl.udel.edu*, *ggao@capsl.udel.edu*, and *tang@capsl.udel.edu*

thread creation, synchronization, and communication. Programmers use these primitives to access the underlying EARTH multithreaded features. A complete reference to the Threaded-C language can be found in [23].

I-structure is a *non-strict* data structure proposed as an extension to the functional language Id by Arvind and his colleagues [3]. This data structure can be used as a synchronization mechanism to support producer and consumer type of computation. Because an I-structure is able to queue read operations when they arrive before the corresponding write operation, the read operation will return the expected value even when it is issued before the write has been performed.

Threaded-C does not provide direct support for I-structures. In this paper we describe the implementation of a library of functions that delivers the functionality of I-structures in Threaded-C. We describe a parallel programming kernel based on a Hopfield Network and present our implementation of this kernel in Threaded-C using I-structures. This implementation demonstrates how I-structures facilitate the job of the programmer to synchronize readers and writers.

We include a brief description of the architecture in section II, a good description of the Threaded-C language can be found in [23]. We present a brief description of I-structures in section III and present our implementation of I-structures in Threaded-C in section IV. Section V describes a parallel kernel based on Hopfield Networks, and Section VI describes our implementation of this kernel in Threaded-C using I-structures. In Section VII we discuss related work.

II. The EARTH Architecture

In the EARTH programming model, threads are sequences of instructions belonging to an enclosing function. Threads always run to completion – they are non-preemptive. Synchronization mechanisms are used to determine when threads become executable (or ready). Although it is possible to spawn a thread explicitly, in most cases a thread starts executing when a specified *synchronization slot* counter reaches zero. A synchronization slot counter is decremented each time a synchronization signal is received. In a typical program, such a signal is received when some data becomes available. Besides the counter, a synchronization slot holds the identification number, or *thread id*, of the thread that is to be started when the counter reaches zero. This mechanism permits the implementation of dataflow-like firing rules for threads (a thread is enabled as soon as all data it will use is available).

An EARTH computer consists of a set of EARTH nodes connected by a communications network.[1] Each EARTH node has an *Execution Unit* (EU) and a *Synchronization Unit* (SU) linked to each other by queues (see Figure 1). The EU executes active threads, and the SU handles the synchronization and scheduling of threads and communication with remote processors.

This division allows the implementation of multithreading architectures with off-the-shelf microprocessors mass-produced for uniprocessor workstations [18]. The EU is expected to be a conventional microprocessor executing threads sequentially.[2] The SU performs specialized tasks and is relatively

[1] The EARTH model does not specify the network's topology.

[2] More precisely, the EU executes threads according to their *sequential semantics*. Naturally, such a processor could take advantage of conventional techniques for speeding up sequential threads, such as out-of-order execution and branch prediction.

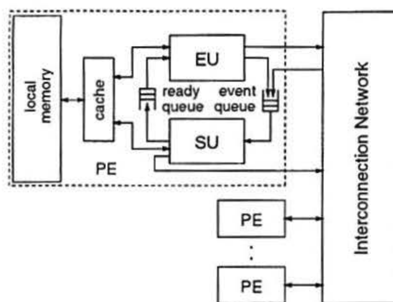

Fig. 1. EARTH architecture

simple compared to the EU. Thus, the SU can be implemented in a small ASIC chip. The two queues connecting the EU and SU may be in separate hardware or may be part of the EU and/or SU.

The function of the queues shown in Figure 1 is to buffer the communication between the EU and SU. The *ready queue*, written by the SU and read by the EU, contains a set of threads which are ready to be executed. The EU fetches a thread from the ready queue whenever the EU is ready to begin executing a new thread. The *event queue*, written by the EU and read by the SU, contains requests for synchronization events and remote memory accesses, generated by the EU. The SU reads and processes these requests as fast as it is able. Request from the EU for remote data can go directly to the network or go through the local SU; implementation constraints will determine the best mechanism, so this is not defined in the model.

To assure flexibility, the EARTH model does not specify a particular instruction set. Instead, ordinary arithmetic and memory operations use whatever instructions are native to the processor(s) serving as the EU. The EARTH model specifies a set of EARTH operations for synchronization and communication. These operations are mapped to native EU instructions according to the needs of the specific architecture. For instance, on a machine with ASIC SU chips, the EU EARTH instructions would most likely be converted to loads and stores from/to memory-mapped addresses which would be recognized and intercepted by the SU hardware.

To maximize portability, the EARTH model makes minimal assumptions about memory addressing and sharing. An EARTH multiprocessor is assumed to be a distributed memory machine in which the local memories combine to form a global address space. Any node can specify any address in this global space. However, a node cannot read or write a non-local address directly. Remote addresses are accessed with special EARTH operations for remote access. A remote load is a *split-phase transaction* with two phases: *issuing the operation* and *using the value returned.* The second phase is performed in another thread, after the load has completed.

III. I-STRUCTURES

I-structures are data structures introduced by Arvind and his collaborators in the context of the functional programming language Id [3]. The most salient advantage of an I-structure is that there is no need for synchronization between reads and writes at their issuing time. An I-structure is considered to be an array of elements[3], where each element of the array can be in one of three states: *empty*, *initialized*, and *suspended*. Each element of the array can only be written once, but it can be read many times. Right after allocation all the elements of the array are in the *empty* state. Conceptually, if a read occurs before the write the element goes into the *suspended* state and the read operation is kept in a local queue. Subsequent reads are also queued. When a write occurs, if the element been written is in the *empty* state, the value is written in the array and the element goes into the *initialized* state. If the element was in the *suspended* state, all the reads that were queued for that element are serviced before the writing operation is complete, and the element goes into the *initialized* state. A read to an *initialized* element returns immediately with the value previously written. A write to an element that is in the *initialized* state is considered a *fatal error* and causes the program to terminate.

In this document we present a set of functions that implement the functionality of I-structures in Portable Threaded-C (PTC). Functional language environments have a mechanism called *garbage collector* that is responsible for reclaiming the memory previously allocated for I-structures that are no longer needed. In languages such as Threaded-C this mechanism is not available. Therefore we need to introduce two new operations that were not part of the original I-structure proposition: *delete* and *reset*.

Observe that for the proper functioning of an I-structure, the *read* and *write* operations must be atomic. This implementation of I-structures in PTC running on the existing EARTH platforms derives atomicity from two assumptions: threads are non-preemptive; and only a single thread can run on a node at a time. The first condition is inherent to the EARTH model, the second condition might not be valid in future implementations (in SMP clusters for example). However both conditions are true for all the current implementations of EARTH systems.

IV. IMPLEMENTING I-STRUCTURES IN THREADED-C

This document describes the support for *I-structures* implemented in Portable Threaded-C through a set of library functions. In this implementation an I-structure can be allocated from the heap and when it is no longer in use can be returned to the heap. The library supports five operations in an I-structure: *allocate*, *read*, *write*, *reset*, and *delete*.

The implementation of *I-structures* described in this document implements the state transition diagram displayed in Figure 2. The states in this figure represent the state of individual elements within an I-structure. The operations in the state transition can be separated in two groups. Operations that cause all the elements of the array to change state: *allocate*, *reset* and *delete*; and operations that

[3] I-structures were defined as arrays of elements in the seminal work of Arvind, Nikhil, and Pingali [3]. However, nothing prevents the implementation of single element i-structure, or other data structure organizations.

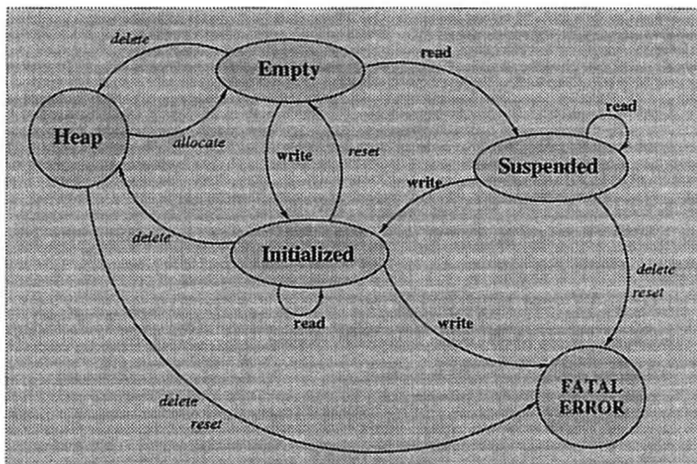

Fig. 2. State Transition Diagram for the I-Structure Implementation

cause a single element of the array to change state: *read* and *write*. We consider that previous to its allocation an I-structure is in the memory heap. When an I-structure is allocated all its elements are placed in the *empty* state. As read and write operations are performed, individual elements can then be moved to the *suspended* or *initialized* state. If any element goes into the *FATAL ERROR* state, the program emits an error message and terminates. Only three conditions are cause for a fatal error in this implementation:

- a write to an array element that has already been initialized;
- a delete or a reset of an I-structure that contains at least one element in the suspended state;
- a delete or a reset of an I-structure that is in the heap (was never allocated or has already been deleted);

Observe that there are other situations that will cause error but that are not checked in this implementation, such as a write or a read to a non-allocated I-structure or a write or a read out of the bounds of the allocated I-structure. The functions that implement I-structure in Threaded-C are listed in Table I.

V. The Hopfield Kernel

In this section we introduce a kernel, based on the Hopfield Network, to illustrate the utilization of the I-structures presented in this document. The motivation for the introduction of this kernel is to illustrate the use of I-structures to provide for simpler user defined synchronization in multithreaded programming.

The Hopfield Network is a recursive neural network that is often used in combinatorial optimization problems and as an associative memory [11]. In both cases the network is formed by a set of neurons

THREADED I_INIT(SPTR slot_adr)
THREADED I_ALLOCATE(int array_length, void *GLOBAL *GLOBAL place, SPTR slot_adr)
THREADED I_READ_x(void *GLOBAL array, int index, int *GLOBAL place, SPTR slot_adr)
THREADED I_READ_BLOCK(void *GLOBAL g_array, int index, long block_size, void *GLOBAL place, SPTR slot_adr)
THREADED I_WRITE_x(void *GLOBAL array, int index, T value)
THREADED I_WRITE_BLOCK_SYNC(void *GLOBAL array, int index, long block_size, void *GLOBAL origin, SPTR slot_adr)
THREADED I_DELETE(void *GLOBAL array)
THREADED I_DELETE_BLOCK(void *GLOBAL array)
THREADED I_RESET(void *GLOBAL array)
THREADED I_RESET_BLOCK(void *GLOBAL array)

TABLE I

LIBRARY OF FUNCTIONS THAT IMPLEMENT I-STRUCTURES IN THREADED-C.

that are connected by synapses[4]. Every neuron is connected to every other neuron in the network.

For the kernel that we introduce in this section we consider only the situation in which the network is placed in a given initial unstable state and is allowed to move to a stable state. In this case the values of the synapses are fixed and the only values that change are the values of the activation level of the neurons. In this *recollection* mode, the value of the output of each neuron at time $k+1$ is given by the following equation.

$$S_j(k+1) = \text{sgn}\left[\sum_{i=1}^{N} w_{ji} S_i(k)\right] \quad (1)$$

Where w_{ji} is the weight of the synapse connecting neuron i to neuron j, $S_i(k)$ is the output of neuron i at time k, and sgn() is the sign function that evaluates to $+1$ if its argument is positive and evaluates to -1 if its argument is negative. In order to update its activation level, a neuron computes the sum of the product of each one of its synapses and the output level of the corresponding neuron.

VI. IMPLEMENTING THE HOPFIELD KERNEL IN THREADED-C

The Hopfield Network is a recursive neural network that is often used in combinatorial optimization problems and as an associative memory [11]. In both cases the network is formed by a set of neurons that are connected by synapses[5]. Every neuron is connected to every other neuron in the network.

Figure 3 presents the structure of our implementation of the Hopfield network in Threaded-C using

[4]In this paper we discuss a Hopfield kernel suitable for the implementation of an associative memory. With few modifications a similar kernel for the resolution of combinatorial optimization problems can be implemented.

[5]In this paper we discuss a Hopfield kernel suitable for the implementation of an associative memory. With few modifications a similar kernel for the resolution of combinatorial optimization problems can be implemented.

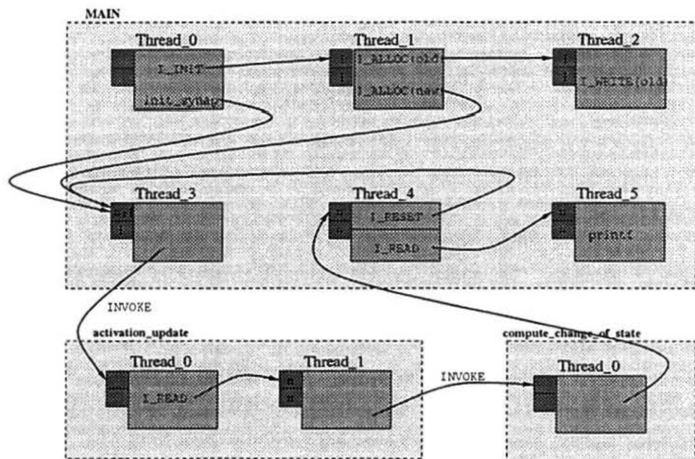

Fig. 3. Synchronization Structure of the Hopfield Kernel Implementation

I-structures[6]. The program has three functions: `MAIN` with six threads, `activation_update` with two threads, and `compute_change` with a single thread. In this section we introduce the code and explain each piece of the program. Observe in the figure that most of the computation is spent in the loop formed by threads 3 and 4 of the main function and by the function `activation_update` and `compute_change`. Threads 0, 1, and 2 of the main function are necessary for initialization and thread 5 prints the final results. In Figure 3 each thread is annotated with the initial count and the reset count of the sync slot that causes the thread to be spawned.

Because of space limitation, we do not provide neither the I-structure implementation, nor the Hopfield network kernel code in this paper. Interested readers can fetch the files with the code from *http://www.capsl.udel.edu/DOCUMENTS/i-struct.tar.gz.*

In Threaded-C each node has its own copy of a global scoped variable. Our implementation uses global scoped variables to store the values of the synapse weights as arrays of floats. Each EARTH node is in charge of computing the new activation level of one neuron. An invocation of the function `I_INIT()` must precede the invocation of any other I-structure in that node. We use two I-structures in this implementation, one to contain the values of the activation in the previous iteration, and another to contain the new activation values. At the end of each iteration, pointer swaping is used to transfer the new values to the old i-structure, preventing expensive copying. The use of i-structures relieves the programmer of some complex synchronization responsibilities.

At the end of each iteration the amount of change in the state of the network is computed in node 0. When this change is below an established threshold, we consider that the network has converged to

[6] A complete description of the Threaded-C language can be found in [23], and a description of the set of functions that implement the I-structure mechanism is described in [2].

an stable state.

VII. Related Work

The Hopfield network was introduced by John Hopfield in a seminal 1982 paper [12]. Its application as an associative memory as well as to find solutions for combinatory optimization problems have since been widely studied [14], [13].

EARTH is a fine grain multithreaded architecture originally developed to be implemented in a distributed memory platform. Currently research at the University of Delaware is been performed to develop an implementation of the EARTH architecture on a cluster of shared memory machines. Another important fine grain multithreaded architecture is the MIT Cilk architecture [5], [10]. In Cilk the distribution of threads among distinct processors is performed through a mechanism called *work stealing*: a new thread is always spawned in the local processor. Whenever a processor runs out of work it tries to steal threads from processors that have more threads than what they can process. To the best of our knowledge, non-strict data structures, such as I-structures, have not been implemented in Cilk.

After the original proposition of I-structures, Arvind and his collaborators proposed M-structures[4]. The main difference between an M-structure and an I-structure is that in an M-structure a read operation — which is called *take* — resets the location to the empty state. When a write operation — called *put* — is performed to a location that has more than one *take* waiting, only one of the *takes* is served and the location remains empty. The advantage of an M-structure is that it allows for multiple writes to the same location. Its disadvantage is that it only allows a single read for each value written. M-structures can be also implemented as a library of functions in Threaded-C in a similar fashion as the implementation of I-structures described in this paper. I-structures are included as instructions in the pH language [1], a parallel dialect of Haskell [15].

An interesting research question is whether and how data structures such I-structures and M-structures could benefit from caching. Dennis and Gao propose four possible approaches to implement a *defer queue* where read operations that cannot be immediately serviced can be stored [8], [9]. They choose to store in memory a list of identifiers of the processor nodes that have one or more pending reads for a memory location. Each processors itself holds a list of continuations for the requested read operation.

VIII. Conclusion

In this paper we proposed a new parallel kernel based on the Hopfield network and described the implementation of I-structures as a library of functions on Threaded-C. The EARTH architecture has been implemented in different platforms, including SP2, MANNA and Beowulf [6], [7], [20]. In each one of these platform the ratio between communication costs and processing costs are different. At the time of the submission of this paper we are working on experiments to investigate how this different ratios affect the performance of our implementation of the Hopfield kernel in EARTH using I-structures. We

are also working in an implementation of the Hopfield kernel that does not use I-structures to study the effect of using I-structures for synchronization. We intend to present the numerical results of our studies at the conference in September.

Acknowledgments

The authors acknowledge the support from DARPA, NSA, NSF (MIPS-9707125), and NASA. Section II is due to Kevin Theobald. The authors also would like to thank all the members of the HTMT team in Delaware — Thomas Geiger, Gerd Heber, Cheng Li, Andres Marquez, Kevin Theobald, Parimala Thulasiram, and Ruppa Thulasiram — for the productive discussions on the implementation of I-structures in our weekly meetings. Special thanks to Elif Albuz for carefully reading the first draft and to Gerd Heber for his patience and time helping the authors figure out some of the initial bugs in the program.

References

[1] S. Aditya, Arvind, and J.-W. Maessen. Semantics of ph: A parallel dialect of haskell. Technical Report CSG-Memo-369, Massachusetts Institute of Technology, Cambridge, MA, June 1995. Published at FPCA'95 Conference. (http://csg-www.lcs.mit.edu:8001/cgi-bin/search.pl?author=arvind).

[2] J. N. Amaral. Implementation of i-structures as a library of functions in portable threaded-c. Technical Report CAPSL TN04, University of Delaware, Newark, DE, June 1998.

[3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[4] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. Technical Report CSG-327, Massachusetts Institute of Technology, Cambridge, MA, March 1991. also appear in *Proceedings of Functional Programming and Computer Architecture*, Cambrideg, MA, August, 1991. (http://csg-www.lcs.mit.edu:8001/cgi-bin/search.pl?author=arvind).

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 19–21, 1995. *SIGPLAN Notices*, 30(8), August 1995.

[6] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium* [19], pages 704–709.

[7] Haiying Cai. Dynamic load balancing on the EARTH-SP system. Master's thesis, McGill University, Montréal, Québec, May 1997.

[8] J. B. Dennis and G. R. Gao. Memory models and cache management for a multithreaded program execution model. Technical Report CSG-363, Massachusetts Institute of Technology, Cambridge, MA, October 1994.

[9] Jack B. Dennis and Guang R. Gao. On memory models and cache management for shared-memory multiprocessors. ACAPS Technical Memo 90, School of Computer Science, McGill University, Montréal, Québec, December 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[11] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan College Publishing Company, New York, NY, 1994.

[12] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings National Academy of Science*, 79:2554–2558, Apr. 1982.

[13] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[14] J. J. Hopfield and D. W. Tank. Simple neural optimization networks: An a/d converter, signal decision circuit, and a linear programming circuit. *IEEE Transactions on Circuits and Systems*, CAS-33(5):533–541, May 1986.

[15] P. Hudak, S. P. Jones, and P. Wadler, editors. *Report on the Programming Language Haskell: A Non-strict Purely Functional Language*, volume 27 of *ACM Sigplan Notices*, May 1992. Version 1.2.

[16] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

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

[18] 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* [19], pages 288–294.

[19] IEEE Computer Society. *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, April 26–29, 1994.

[20] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News*, 24(2), May 1996.

[21] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf:harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of IEEE Aerospace*, 1997. http://cesdis.gsfc.nasa.gov/beowulf/papers/papers.html.

[22] T. Sterling, D. J. Becker, D. Savarese, M. Berry, and C. Res. Achieving a balanced low-cost architecture for mass storage management through multiple fast ethernet channels on the beowulf parallel workstation. In *Proceedings of the International Parallel Processing Symposium*, 1996. http://cesdis.gsfc.nasa.gov/beowulf/papers/papers.html.

[23] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the portable threaded-c language. CAPSL Technical Memo 19, University of Delaware, http://www.capsl.udel.edu, April 1998.