A Survey of Load Balancers in Modern Multi-Threading Systems

Prasad Kakulavarapu[1], José Nelson Amaral[2]

[1] McGill University
School of Computer Science
Montréal, Canada
prasad@cs.mcgill.ca
[2] University of Delaware
CAPS Lab, Dept. of Electrical and Computer Engineering
Newark, USA
amaral@capsl.udel.edu

Abstract— **Multithreaded architectures are a feasible approach to exploit both regular and irregular parallelism. Today a large collection of multi-threading architectures with different threaded models, ad implementation platforms are available. These architectures provide support for multithreading either at hardware level, with customized functional units, or at the software level, as emulators written in some high-level language. The later approach is usually preferred because of its favorable price tag, speed of development, and portability. In this article we review some of these architectures focusing in their capabilities to provide load balancing for irregular, data-parallel and recursive applications. The paper is anchored on a description of our own implementation of load balancers for EARTH - Efficient Architecture for Running THreads. Most of the multithreading architectures that we review are software emulations based on off-the-shelf hardware and compiler technologies. In a related paper we detail the implementation of the EARTH runtime system.**

Keywords— **Fine-grain parallelism, multithreading, non-blocking threads, context-switching, runtime system, distributed memory, dynamic load balancing.**

I. Introduction

In the classical strict data-flow model of computation, an instruction is enabled for execution when all its operands are available [13], [17], [12], [24]. To enforce the enabling condition, the instructions that produce such operands must be able to send a synchronization signal to all the instructions that will consume the recently produced result. This model proved unyielding for the implementation of machines based on current standard off-the-shelf hardware and compiler technology. However many research groups have successfully implemented a model of computation that is a direct evolution of the classical data-flow model: *fine grain multi-threading*. In the later, the unit of computation is no longer an instruction, but a code-block formed by many instructions. An instantiation of the code-block running on a processing node is called a *thread*, thus the name multi-threading for these systems. Threads, and not individual instructions, are enabled by synchronization signals. The main motivation for the design of multi-threading system is the overlapping of communication and synchronization latencies with computation.

Around the same time that architectures derived from the data-flow model were proposed, the term *thread* started to be used to refer to multiple contexts of computation in operating systems. These threads represent different lines of control that are active at the same time within an OS process. We refer to such threads as *OS-threads*. Well known OS-thread systems include POSIX Threads, Solaris Threads, and NT Threads. OS-threads share all the resources of a process such as memory space, files, and device drivers. However, each thread has its own set of registers, and its own stack, which are usually stored in heap memory. Context-switching between these threads is far easier than that between processes, as there is no need to save and restore memory pointers and other process related resources. Only the contents of the thread specific stack and register set need to be swapped at context-switch time. Programming applications at the level of these threads, rather than at the process level is advantageous because of the high-speed context-switching among threads.

There is a major historical difference between the fine grain threads discussed earlier and OS-threads. Fine grain threads are generated from code-blocks that grow upwards from the data-flow single instruction. A fine grain thread is the largest unit of code that can run without incurring any long latencies due to dependence on other pieces of code or on data stored remotely. OS-threads grow downward from the process abstraction in operating system. An OS-thread is the smallest segment of code that can share a set of resources with the other threads of the same process. Typically OS-threads exploit parallelism at a coarser grain than fine grain threads, and thus must execute a higher number of instructions between thread switchings.

In the multi-threading systems that we discuss in

this paper, each processing unit issues instructions from a single thread at any time [1]. An alternative multi-threading system is called *simultaneous multi-threading* (SMT). In an SMT system a single processor is capable of issuing instructions from multiple threads simultaneously [9]. Machines with such an organization use multiple threads of computation to hide the latency incurred due to the fetching of data from the local memory. An example of the later is the Tera machine [1].

All the platforms discussed in this paper fall in the category of *distributed multi-threading* systems. They are implemented on clusters of off the shelf computers and use threads of computation to hide the latency of fetching data from remote regions of the memory, most likely in the memory of another processing node. These platforms do not use multi-threading to hide the latency caused by a cache miss, i.e., as long as the memory address referenced is in the memory hierarchy of the local processing node, the reference is regarded as a local access.

This paper is organized as follows. Section II reviews preemptive, cooperative, blocking, and non-blocking thread models. Section III categorize modern implementations of multi-threading system in language-based and library-based systems. In section III-A we present an extended discussion of EARTH, Cilk, and TAM, three multi-threading system with extensive effort on language support. In section III-E we review many multi-threading systems whose implementation is based on function libraries and that rely on OS-threads.

II. Threading Models

Fine-grain multi-threading architectures might be characterized by their threading model. Threads can adopt the *cooperative* multithreading model, where threads voluntarily release the CPU, or the *preemptive* model where threads can utilize the CPU only as long as certain conditions specified by the scheduler are valid. Cooperative threads can be *non-blocking* or *blocking*. In a non-blocking system, threads must run until completion. Under a blocking threading model a thread can block when an operation with long or unpredictable latency is encountered in the application. In this case the machine state has to be saved to be restored later. In a preemptive threading model, the scheduler determines the running time of a thread based on its scheduling policy, which may be based on priority, time-slices or a combination of both. In a preemptive threading system, threads are always blocking, and threads enter the blocked state either due to an operation in the program or due to a scheduling decision.

[1]When these systems are implemented on top of super-scalar/super-pipelined processors multiple instructions belonging to the same thread can be issued at one time.

In a non-blocking and non-preemptive thread model, operations with long or unpredictable latencies must be executed in a *split-phase fashion*. The first phase of the operation, also referred to as the *issuing* of the operation is performed in one thread, while the second phase, sometimes referred to as the *consumption* of the result of the operation is performed in another thread. When such a thread model is chosen, a mechanism must be provided to enable the issuing thread to specify which one is the consuming thread. There is no need to preserve machine state during context-switch time.

Neither cooperative blocking thread model nor a preemptive threading model are very attractive for fine-grain multi-threading architectures because the removal of the context of a thread from the processing unit requires that the contents of the registers and the stack must be saved in a temporary user-area before context-switching, and these must be reloaded again when the suspended threads are enabled at a latter time. In addition, this model might be unyielding for the implementation of machine-independent multi-threaded platforms. Also dynamic and irregular applications might cause excessive waste of cycles when mapped to a blocking thread model.

III. Implementations of Multi-threading Platforms

The multi-thread systems that we discuss in this paper are software emulations of architectures. Most of these emulations are based on off-the-shelf hardware and compiler technology. These systems can be broadly divided in two classes.

Language-Based Systems: These systems are based on the support of a custom runtime system. The runtime system implements an interface with the hardware and the system level software in the machine and provides a standard interface for portable implementations of the multi-threading program environment. These systems often offer a language with multi-threaded constructs, and a source-to-source translator to convert this language to a standard and broadly supported language, such as `C`. The advantage of these systems is that threads are usually non-blocking and execute in user space. Thus overheads associated with thread switching are reduced, resulting in very light-weight threads. These systems can be implemented efficiently in both shared and distributed memory platforms. Examples of systems in this class include EARTH [16], [20], [15], [24], [18], [14], Cilk [11], and TAM [8].

Library-Based Systems: These systems provide a library of multi-threaded primitives to manage user level threads on top of OS threads. In this approach

the management of threads requires a few system calls, which is costly in terms of execution cycles. Most of the thread library packages that we found in the literature are designed for shared memory or distributed shared memory systems. One exception is the Chant library [22] that extends the POSIX standard for light-weight threads with functionality for distributed memory environments. Examples of systems based on library of primitives include Nano-threads [2], Ariadne [21], Opus [22], Structure Thread Library [25], and Active Threads [27].

A. Language-Based Systems

In this section we present three fine grain multithreading systems. Each of these systems supports non-blocking, non-preemptive threads. First we describe our own home-grown EARTH system. The development of EARTH started at the McGill University in Montreal, Canada, and continues at the University of Delaware, USA. The original inspiration for EARTH has been derived from the McGill Dataflow Machine [13]. The research around EARTH has spawned over many fields including the development of pre-processors, runtime systems, language development, application studies, source-to-source compilers, and dynamic load balancers. Recently an evolutionary path for the EARTH system was envisioned chartering the progressive development of further customized platforms [24]. The EARTH system has been implemented on the MANNA machine, IBM SP2, Beowulf and on a SUN SMP cluster.

Leiserson *et al.* at MIT developed Cilk, an algorithmic multi-threaded language currently designed for symmetric multiprocessors (SMP's). Central to Cilk's development is the scheduling of multi-threaded computations using a work-stealing mechanism. The Cilk computation model and its implementation are described in [5]. Earlier releases of Cilk implement the memory model called "dag consistency" [4]. Cilk is a succinct extension to C and has the "C elision property": when all the Cilk constructs are removed from a Cilk code, what remains is a legal C code. The most recent release of Cilk is described in [11]. The Cilk group is well known for their implementation of world-class chess programs on the Cilk platform. A unique feature of Cilk is the development of a novel debugging tool, called "Nondeterminator", that finds data races in the execution of programs [7].

The Threaded Abstract Machine project [8] at the University of Berkeley, California presents an execution model in which the compiler controls the synchronization, scheduling and storage management. The role of the compiler in scheduling and management of threads is emphasized to take advantage of critical processor resources such as register storage and exploit considerable inter-thread locality. TAM was one of the first multithreaded systems that were built through software emulation with minimal hardware support. The compiler translates programs written in the functional language *Id* into an intermediate language called TL0, which includes code generated for thread support [23] in a distributed memory environment. An important feature in TAM is the introduction of *inlets* which are specialized message handlers to support inter-frame communications. These inlets are generated by the compiler, one for every value to be received.

B. The EARTH Model

A *thread* in EARTH is a set of instructions that are executed sequentially. Interacting threads sharing context are grouped into *threaded functions*, and are represented in the EARTH runtime system as *tokens*. Applications execute in global memory space comprising the local memories on all the nodes in the system. Applications in EARTH are written in Threaded-C, a multithreaded variant of C. Fig. 1 shows a typical activation graph for a Threaded-C program.

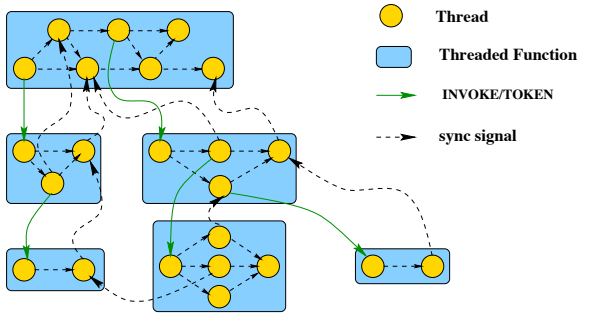

Fig. 1. A generic activation graph for a Threaded-C program.

Three important features characterize the EARTH model [16], [24]:

Synchronization Slots: Conceptually, each processing node has a table of synchronization slots. Any threaded function can allocate a slot, initialize its counter and its reset value, associate the slot `n` to a thread, and pass the address of the slot to other thread functions. Synchronization signals are sent to slots. Each arriving signal causes the slot counter to be decremented. When the counter reaches zero the associated thread is enabled for execution and the counter is reset to the specified reset value. The versatility of the synchronization slots allows for the construction of generic call graphs, such as the one illustrated in Figure 1.

Synchronization Unit: The EARTH model assumes that a functional unit is provided to implement communication, synchronization, and dynamic load balancing functions. The functions of

the SU can be implemented by a second processor in the processing node, by custom hardware, or it can be emulated in software when the EARTH system is implemented on clusters of off the shelf computers, such as the IBM-SP2 and the Beowulf implementations.

Dynamic Load Balancer: Balancing the work load for irregular and data-parallel applications in fine-grain multi-threading architectures might be challenging. Seven distinct dynamic load balancing algorithms have been implemented for EARTH and their performance is studied [6]. Central to the implementation of EARTH's load balancers is the instantiation of threads as migratable tokens, and the implementation of a storage mechanism that behaves as a stack operatable on both ends, as illustrated in Figure 2 [18], [6]. Locality is favored with this mechanism, because tokens generated locally are more likely to be executed in the local processing unit while tokens that arrive from other nodes are more likely to migrate.

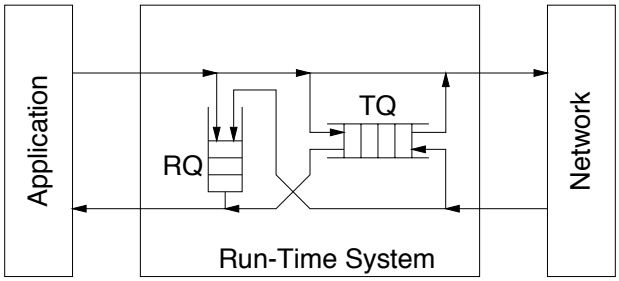

Fig. 2. Internal Queues in the EARTH Runtime System

EARTH has dynamic load balancers tailored for fine-grain multi-threading. The balancers aim to ensure that all nodes are busy, rather than trying to distribute the workload equally among all the nodes in the system. Three kinds of balancers are implemented: receiver-initiated, sender-initiated and hybrid balancers. Significant performance gains have been obtained with load balancing [6]. The results have also demonstrated the difficulty in designing a single load balancer that is perfect for all applications. As an important result of this study, hybrid balancers that rely on history information have been suggested for best performance and scalability in applications representing irregular, structured and recursive parallelism.

C. The Cilk Multi-threaded Language

The Cilk multi-threaded language [11] is an extension to C, and processes user-level fine-grain, non-blocking threads in a shared memory environment. The Cilk compiler generates two versions of target C code for each Cilk procedure - a fast clone and a slow clone. The fast clones are meant for local execution of a procedure, and the slow clones are used as units for dynamic load balancing. The Cilk runtime system [5] employs a randomizing, work-stealing scheduler and operates on a double-ended queue that is similar to the token queue in the EARTH runtime system [16]. Such queuing structure was developed earlier in the ADAM architecture [19].

The Cilk threading model is very amenable for the solution of divide-and-conquer problems, and is most suited for fully-strict computations [5]. While the directed-acyclic graph formed from a Cilk multi-threaded computation allows communications between parent and child procedures, it does not support communications between threads belonging to different Cilk procedures that are at the same level in the activation graph. In contrast, the EARTH threaded model enables the implementation of any arbitrary activation graph through the exchange of synchronization slot addresses. The efficiency of the Cilk scheduler is analytically studied [3].

D. The Threaded Abstract Machine

A TAM program is a collection of *code-blocks*, similar to EARTH programs which are collections of threaded functions [8]. Each code-block, like a threaded function in EARTH, consists of several threads. However, a code-block also includes code for the inlets. Since an activation frame corresponding to a code-block is allocated on a processor, all the threads belonging to a code-block execute on the same processor. For this reason, code-blocks are the units of workload rather than individual threads, as is the case of threaded functions in EARTH. However the distribution of this workload onto the processors in the system is decided by the TAM compiler [23], whereas in EARTH the workload is dynamically distributed at runtime by the load balancer.

A *quantum* in TAM is the number of threads belonging to a code-block that are enabled for execution at any particular instant of time. All the threads in a quantum are executed consecutively, and values defined and used within a thread can be retained in processor registers. This is unlike EARTH, where enabled threads belonging to different threaded functions are placed in a FIFO ready queue, and therefore threads from different threaded functions execute on a first-come basis. In EARTH, threads in a threaded function usually have synchronization dependences between them. Therefore, it is highly unlikely that there many threads of the same threaded function are enabled at the same time to take advantage of TAM's register usage technique. Further, the gains from register usage as in TAM may be insignificant when there is a single or a few enabled threads in a quantum. Another difference between EARTH and

TAM is the dynamic scheduling of threads. In EARTH, the ready queue (FIFO) and the token queue (DEQUE) are used for local and remote scheduling of threads, whereas complex entry and exit codes have to be generated for each quantum by the compiler in TAM.

E. Library-Based Systems

In this section we present multi-threaded systems that are implemented on top of operating system based threads. Although such systems might be more portable because they can run in any machine that supports the underlying operating system, they pay a high price on the cost of system calls to implement thread switching.

F. Distributed Filaments

The distributed Filaments system [10] offer multi-threaded primitives to implement fine-grain threads in a distributed shared memory model. The Filaments runtime system implements distributed shared memory with no hardware support over distributed memory systems. The threads are blocking in nature, and favor irregular, data-parallel and recursive applications. There are multiple server threads per-node, and each server thread executes a set of sharing context filaments (called a pool). In the case of irregular and data-parallel threads, the programmer/compiler has to assign context-sharing filaments to pools on different nodes so as to maintain locality and equal task distribution. However, a simple receiver-initiated scheduler distributes workload in the case of recursive threads. This balancer queries other nodes in a round-robin fashion to steal work. A filament blocks when a long latency operation is encountered. Though there is a provision for the programmer/compiler to enable/disable load balancing in Filaments, it is difficult to estimate runtime load imbalances at compile-time, especially in the case of fine-grain applications.

G. The Opus Language

The Opus language [22] provides Fortran language extensions to support task and data parallelism. Independent tasks representing coarse-grain parallelism, communicate and synchronize through monitor-like structures called shared-data-abstractions. The Opus runtime system relies on a light-weight threads package called Chant, to support multithreading functionality in a distributed memory environment. The Chant threads package extends the pthreads interface with primitives for remote communications, remote thread operations by using existing communication library (MPI standard). Workload has to be mapped onto different nodes by the programmer/compiler keeping in mind locality of the tasks as there is no runtime dynamic load balancing support.

H. Nano-Threads

The Nano-Threads [2] are user-level threads built on top of kernel threads. The Nano-threads library provides primitives to support multithreading efficiently in a multi-user/multiprocessor environment with shared memory. A compiler takes as input C/Fortran programs with Nano-Threads keywords, and generates target C/Fortran code (Nano-Threads) along with code to manage an intermediate representation of varying levels of parallelism in the application, called the Hierarchical Task Graph. The associated code chooses the appropriate granularity for execution at runtime, depending on the availability of resources. Each Nano-Thread is associated with a per-thread-counter and a nano-thread descriptor. Nano-Threads block so that child threads can access local variables from the address space of the parent nano-thread. All enabled Nano-threads are placed in globally accessible and manageable ready queue called GQ (FIFO). To preserve locality, each node has its own local queue (FIFO) that is accessible from all nodes. The objective of load balancing in the Nano-Threads system is to distribute the load equally among all the nodes. This is a different goal from the one adopted on EARTH, where the aim is to keep all processors busy, thereby minimizing balancer overheads in an extremely fine-grain environment. Another potential balancing overhead may be the contention problems for controlling the global queue which may degrade scalability of the system.

I. Active Threads

The Active threads library [27] define an interface for supporting fine-grain, non-preemptive, blocking threads over traditional kernel threads. They can be used to hand code applications, or as virtual machine target for compilers of parallel languages. Threads sharing context are grouped into bundles. Each bundle has its own scheduler and the scheduler may be chosen by the application from a set of schedulers distributed with the active threads package. The scheduler maps active threads onto processor thread dispatch buffers for each processor. Though the fast threading primitives ensure low overheads for thread operations, the multi-threading overheads for thread initialization, context-switching, thread stack management and synchronization are quite high for irregular applications employing fine-grain threads. In contrast, context-switching in EARTH is as cheap as a `C` function call, and there is no need for thread stack management.

J. Concert, Structured Threads and Ariadne

The Concert runtime system [26] proposes close coupling with the compiler and hardware to overcome overheads associated with thread management and commu-

nication in a distributed memory environment, especially when dealing with fine-grain threads for dynamic and irregular applications. The hybrid stack-heap execution mechanism overcomes multithreading overheads, and the pull-based messaging technique minimizes communication overheads. The structured threads library [25] provides multithreading support on top of kernel threads in Windows NT. Ariadne [21] is a threads library that is modeled for process-oriented parallel and distributed simulations. Ariadne threads run on top of the kernel threads, and are implemented in both shared and distributed memory environments. The internal scheduling policy is based on priority queues, i.e. a highest priority non-blocked thread gets executed first. This library is more suited for coarse-grain parallelism.

References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proc., of Intl. Conf. on Supercomputing, Amsterdam, The Netherlands*, pages 1–6, June 1990.

[2] Eduard Ayguade', Mario Furnari, Maurizio Giordano, Hans-Christian Hoppe, Jesus Labarta, Xavier Martorell, Nacho Navarro, Dimitrios Nikolopoulos, Theodore Papatheodorou, and Eleftherios Polychronopoulos. Nano-Threads: Programming Model Specification. In *Deliverable M1.D1, ESPRIT Project NANOS (No. 21907), University of Patras*, July 1997.

[3] Robert Blumofe and Charles Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. of the 35th Annual Symposium on foundations of Computer Science (FOCS), Santa Fe, New Mexico*, pages 356–368, November 1994.

[4] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 24–26, 1996. SIGACT/SIGARCH.

[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. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[6] Haiying Cai, Olivier Maquelin, Prasad Kakulavarapu, and Guang R. Gao. Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution Model. In *Proc. of the Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida*, January 1999.

[7] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting Data Races in Cilk Programs that Use Locks. In *Proc. of 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98), Puerto Vallarta, Mexico*, pages 298–309, June 1998.

[8] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA*, April 1991.

[9] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors. In *Proc. of IEEE Micro*, pages 12–18, sept 1997.

[10] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proc. of the First Symposium on Operating Systems Design and Implementation, Usenix Association*, November 1994.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[12] Guang R. Gao. An Efficient Hybrid Dataflow Architecture Model. *Journal of Parallelism*, 19(4), December 1993.

[13] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Parallel Function Invocation in a Dynamic Argument-Fetching Dataflow Architecture. In *Proc. of PARBASE-90: Intl. Conf. on Databases, Parallel Architectures, and their Applications, Miami Beach, Florida*, pages 112–116, March 1990.

[14] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the EARTH Multithreaded Architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT'96), Boston, Mass.Intl. Journal of Parallel Programming*, pages 12–23, October 1996.

[15] 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.

[16] 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.

[17] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture.* PhD thesis, McGill University, Montréal, Québec, May 1992.

[18] Prasad Kakulavarapu, Olivier Maquelin, Jose Nelson Amaral, and Guang R. Gao. A Runtime System for Fine-grain Multithreaded Multiprocessor Systems. In *Technical Memo 24, CAPSL, University of Delaware*, may 1999.

[19] Olivier C. Maquelin. Load Balancing and Resource Management in the ADAM Machine. In *Second Workshop on Dataflow Computing, Hamilton Island, Australia, 1992, Published in Advanced Topics in Dataflow Computing and Multithreading, Lubomir Bic, Guang R. Gao, Jean-Luc Gaudiot editors, IEEE Computer Society*, 1995.

[20] Olivier C. Maquelin, Herbert H. J. Hum, , and Guang R. Gao. Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors. In *Proc. of the First International EURO-PAR Conference, No. 966 in Lecture Notes in Computer Science, Stockholm, Sweden*, pages 117–128, August 1995.

[21] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a Portable Threads System supporting Thread Migration. *Software - Practice and Experience*, 26(3):327–356, March 1996.

[22] Piyush Mehrotra and Matthew Haines. An Overview of the Opus Language and Runtime System. Technical Report, Institute for Computer Applications in Science and Eng., NASA Langley Research Center, Hampton, Virginia, May 1994.

[23] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. of FPCA '91 Conference on Functional Programming Languages and Computer Architecture, Springer Verlag*, aug 1991.

[24] Kevin B. Theobald. EARTH - An Efficient Architecture for Running THreads. In *Ph.D Thesis, School of Computer Science, McGill University, Montreal, Québec*, March 1999.

[25] John Thornley, K. Mani Chandy, and Hiroshi Ishii. A System for Structured High-Performance Multithreaded Programming in Windows NT. In *Proc. of the 2nd USENIX Windows NT Symposium, pp. 67-76, Seattle, Washington*, August 1998.

[26] John Plevyak Vijay Karamcheti and Andrew A. Chien. Runtime Mechanisms for Efficient Dynamic Multithreading. *Journal of Parallel and Distributed Computing*, 37:21–40, August 1996.

[27] Boris Weissman. Active Threads: an Extensible and Portable Light-Weight Thread System. Technical Report TR-97-036, Intl. Computer Science Institute, Berkeley, California, September 1997.