A Survey of Load Balancers in Modern Multi-Threading Systems

Prasad Kakulavarapu¹, José Nelson Amaral²

¹ McGill University
School of Computer Science
Montréal, Canada
prasad@cse.mcgill.ca

² University of Delaware
CAPS Lab, Dept. of Electrical and Computer Engineering
Newark, USA
amaral@caps.udel.edu

Abstract—
Multithreaded architectures are a feasible approach to exploit parallelism in both regular and irregular applications. Today a large collection of multi-threading architectures with different threaded models, and 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.

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 [15], [19], [14], [26]. To enforce the enabling condition, the instructions that produce such operands must be available. Because of its favorable price tag, speed of development, and portability, the 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 either stored in heap memory (as in POSIX or Solaris threads) or in kernel space (as in NT threads). 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. In this article, we study multithreaded systems that are implemented on top of OS 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 pa-
per, 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 [11]. 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].

Both shared and distributed memory based platforms are considered in this study. These platforms are implemented with off the shelf computers and use threads of computation to hide latencies associated with either the fetching of data from remote regions of the memory, or synchronizing among other threads. 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 categorizes 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-B we review many multi-threading systems whose implementation is based on function libraries and that rely on OS-threads. Section IV concludes the paper.

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 preserved to be restored later. In a preemptive threading model, the scheduler determines the runtime of a thread based on its scheduling policy. There are two ways to implement this: priority, time-slices or a combination of both. In a preemptive threading system, threads are always blocked, and threads enter the blocked state either due to an operation in the program or due to a scheduling decision.

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 loaded 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-threading 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 interacts 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. Language-based 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 threading model can be determined by the language designer keeping in mind the requirements and semantics of the intended application domain. This design methodology frees the threading model from many constraints or limitations to the multithreading features that could be imposed by the underlying OS platform. The advantage of language-based systems is that threads are usually non-blocking and execute in user space. Thus overheads associated with thread creation and thread switching are reduced, resulting in very lightweight threads. Due to the minimal thread specific storage required, it is possible to have a very high number of threads active at the same time. These systems can be implemented efficiently in both shared and distributed memory platforms. Examples of systems in this class include EARTH [18], [22], [17], [26], [20], [16], Cilk [13], TAM [9] and Athapascan-1 [7].

\(^1\)When these systems are implemented on top of super-scalar/superpipelined processors multiple instructions belonging to the same thread can be issued at one time.
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. In most instances the blocking threaded model is used, which limits the number of simultaneously active threads as the thread context and related data structures have to be retained. 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 [24] 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 [23], Opus [24], Structure Thread Library [27], and Active Threads [29].

A. Language-Based Systems

In this section we present three fine grain multi-threading 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 [15]. 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 [26]. 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 [13]. 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 [8].

The Threaded Abstract Machine project [9] at the University of California, Berkeley 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 ld into an intermediate language called TL0, which includes code generated for thread support [25] 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.

A.1 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.

Three important features characterize the EARTH model [18], [26]:

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 threaded 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 commu-
cation, 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 in [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 operable on both ends, as illustrated in Figure 2 [20], [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](image)

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 performs well 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.

A.2 The Cilk Multi-threaded Language

The Cilk multi-threaded language [13] 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 [18]. Such queuing structure was developed earlier in the ADAM architecture [21].

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 in [3].

A.3 The Threaded Abstract Machine

A TAM program is a collection of code-blocks, similar to EARTH programs which are collections of threaded functions [9]. 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. The distribution of this workload onto the processors in the system is decided by the TAM compiler [25].

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.

An important contribution of TAM is the definition of an abstract machine with a multilevel scheduling hierarchy. This hierarchy created a collection of synchronization mechanisms with different costs and capabilities available for the compiler. The early experiments with TAM demonstrated that significant locality exist in the access pattern of data among the many small threads that the system handles [10]. Another important contribution of TAM is the definition of a program graph and a dual graph to represent a fine grain multi-threaded program [25].

B. 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.
B.1 Distributed Filaments

The distributed Filaments system [12] offer multithreaded 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. Filaments allow the programmer/compiler to enable/disable load balancing.

B.2 The Opus Language

The Opus language [24] 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 lightweight thread package called Chant, to support multithreading functionality in a distributed memory environment. The Chant thread package extends the pthreads interface with primitives for remote communications and remote thread operations by using existing communication library (MPI standard). Workload has to be mapped onto different nodes by the programmer/compiler according to locality constraints of the tasks as there is no runtime dynamic load balancing support.

B.3 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 load balance goal 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.

B.4 Active Threads

The Active threads library [29] 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 multithreading overheads for thread initialization, context-switching, thread stack management and synchronization are quite high for irregular applications employing fine-grain threads.

B.5 Concert, Structured Threads and Ariadne

The Concert runtime system [28] proposes close coupling with the compiler and hardware to overcome overheads associated with thread management and communication 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 [27] provides multithreading support on top of kernel threads in Windows NT. Ariadne [23] is a thread 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 suited for coarse-grain parallelism.

IV. CONCLUSION

This paper studied the threaded model and its implementation, with emphasis on dynamic load balancing in modern multithreaded multiprocessor systems. These various systems are classified on the basis of their runtime system support into language and library based systems. The
blocking/non-blocking nature of threads and their scheduling policies are presented as major yardsticks used to differentiate among the various multithreaded models.

The level-thread levels in language-based systems are more suited to model fine-grain parallelism than those from the library-based systems. This is due to the high thread management and switching costs in the library-based systems. The library-based systems provide high flexibility to the language designer. However, the blocking nature of threads from the library based threads make them ideal to model more real-world applications where the coarser model of the threads overcome the high synchronization and context-switching costs. With the increasing popularity of the SMP clusters, the ideal scenario that is emerging is to support non-blocking threads on top of the kernel threads [12], [2], [27].

V. ACKNOWLEDGMENTS

The authors would like to thank current and former members of the ACAPS Lab, McGill University, and the CAPSL Lab, University of Delaware for their insight, ideas and help. Special thanks are due to Prof. Guang R. Gao, University of Delaware, for the insightful discussions on the design aspects of language-based multithreaded systems. The authors also acknowledge the partial support from DARPA, NSA, NSF (under grants NSF-CISE-9726388 and NSF-MIPS-9707125) and NASA. The initial EARTH work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES


