

# Fine-Grain and Multiprogramming-Conscious Nanothreading with the Solaris Operating System

Dimitrios S. Nikolopoulos   Eleftherios D. Polychronopoulos   Theodore S. Papatheodorou  
High Performance Information Systems Laboratory  
Department of Computer Engineering and Informatics  
University of Patras  
Rion 265 00, Patras, Greece

**Abstract** *This paper presents the architectural and implementation details of a nanothreads runtime system customized for the Solaris operating system. A nanothreads runtime system addresses jointly three major performance issues; exploitation of fine-grain parallelism, efficient execution of arbitrarily nested task and data parallelism and scalability of multithreaded programs in multiprogrammed shared-memory multiprocessors. The nanothreads runtime system in Solaris is implemented as a highly tuned, thin threading layer on top of Solaris threads and exploits effectively the Solaris kernel support for scheduler activations. Nanothreads are implemented with aggressive optimizations and several critical paths in the runtime system are either reduced or eliminated to make the common case fast. Solaris scheduler activations provide a fast communication path between the nanothreads runtime system and the Solaris kernel, which along with non-blocking synchronization enable the scalability of nanothreaded binaries in a multiprogrammed environment.*

**Keywords:** nanothreads, multithreading, multiprocessing/multiprogramming, runtime systems, operating systems.

## 1 Introduction

As modern shared memory multiprocessors move towards larger processor scales and become heavily multiprogrammed, the need for supporting fine-grain parallel processing integrated with multiprogramming becomes more acute. Although several research efforts in the past investigated fine-grain multithreading [4, 5, 17] and the interfer-

ences between multiprocessing and multiprogramming [1, 11], little effort has been paid to address both issues jointly [2, 3]. Furthermore, the adoption of these research proposals in modern operating systems is so far quite limited. Solaris is probably the only vendor operating system which provides a stable user-level threads architecture, coupled with an implementation of scheduler activations [1] to enhance the robustness of parallel programs in the presence of multiprogramming. Although Solaris threads constitute a convenient platform for generic multithreaded programming, raw performance issues make them unattractive for fine-grained parallel programs. On the other hand, the exported functionality of scheduler activations in Solaris is currently limited to preemption-safe locking, although scheduler activations is a very powerful tool for tuning the behavior of parallel programs in a multiprogrammed environment.

This paper presents the architectural and implementation details of a *nanothreads* [12] runtime system in Solaris, for symmetric multiprocessors based on the UltraSPARC-II processor architecture. A nanothreads runtime system addresses jointly three major performance issues; exploitation of fine-grain parallelism; multilevel parallelization and efficient execution of arbitrarily nested task and data parallelism; and scalability of multithreaded programs in multiprogrammed environments. Our work builds on previous research done in multithreading runtime systems for multilevel parallelization on SGI and Intel multiprocessor platforms [5, 7], as well as kernel-level support for multiprogramming scalability of par-

allel programs [3, 12, 14]. To our knowledge, the work presented in this paper contributes the first integrated runtime system for Solaris, which supports efficiently fine-grain and multiprogramming-conscious multilevel parallelism.

The rest of this paper is organized as follows: Section 2 outlines the implementation of the nanothreads runtime system in Solaris. Section 3 presents early performance results of our working prototype on a Sun Enterprise server. Section 4 concludes the paper.

## 2 Implementation

Four major design issues guided the implementation of the nanothreads runtime system in Solaris:

- The cost of creating and executing a thread should be as close as possible to the cost of a function call. Each thread should maintain the absolutely minimal architectural state needed to switch to a different execution flow within the same address space. Moreover, threads state should be maintained only when needed.
- The runtime system should provide a structure for maintaining nested thread contexts with parent-child relationships to enable multilevel parallelization.
- The runtime system should employ a mechanism to poll the state of kernel threads within a parallel program and react to kernel thread preemptions and blocking, which implicitly reflect kernel-side scheduling decisions. Exporting these scheduling decisions to the runtime system is essential to render the program responsible of managing its own kernel threads, in a way that ensures the progress of the program in the presence of unfortunate thread preemptions from the operating system.
- The implementation should exploit as much as possible the vendor's operating system support for multithreading.

We implemented the nanothreads runtime system in Solaris as a very thin threading layer on

top of the Solaris threads library, as shown in Figure 1<sup>1</sup>. This approach eased the integration of nanothreads with the multithreading infrastructure of Solaris and has deemed advantageous, particularly with respect to multiprogramming scalability.

We implemented nanothreads from scratch, using the minimal representation of a thread in the UltraSPARC-II architecture: a heap-allocated stack and the architecture-dependent register file, which consists of 8 *input* registers (six input parameters for the thread, the program counter and the frame pointer), plus 8 *local* registers containing callee-save automatic variables, for a total of 16 registers.

We used non-preemptive threads to reduce thread switching overhead and avoid maintaining extra thread state such as signal handlers and per-thread priorities. Besides, preemptive user-level scheduling is of no practical importance for nanothreaded parallel programs. Put simply, as soon as a program is decomposed into a number of threads that reflects its degree of exploitable parallelism, there is no reason for the user-level scheduler to assign priorities to threads that cooperate to execute the same task and preempt deliberately one thread in favor of another. Among other potential problems, preemptive scheduling may also motivate excessive thread switches. The important exception to the previous rule are idling threads, which may be threads that starve waiting to enter a critical section, or the user-level scheduler threads that do not find any work to execute in the run queues. In the nanothreads runtime system, idling threads employ a self-blocking discipline which is handled in a local scope within the program, in coordination with the kernel. This topic is discussed later in this section.

The trickiest detail of the nanothreads implementation on the UltraSPARC-II was the handling of active register windows [16]. We used the `flushw` instruction of the UltraSPARC-II, which spills register windows with a conditional trap and

---

<sup>1</sup>Solaris uses a three-level multithreading architecture, where parallel programs are decomposed into user-level threads, user-level threads are mapped to execution vehicles called lightweight processes (LWPs), LWPs are mapped one-to-one to kernel threads and kernel threads are finally mapped to physical processors by the operating system.

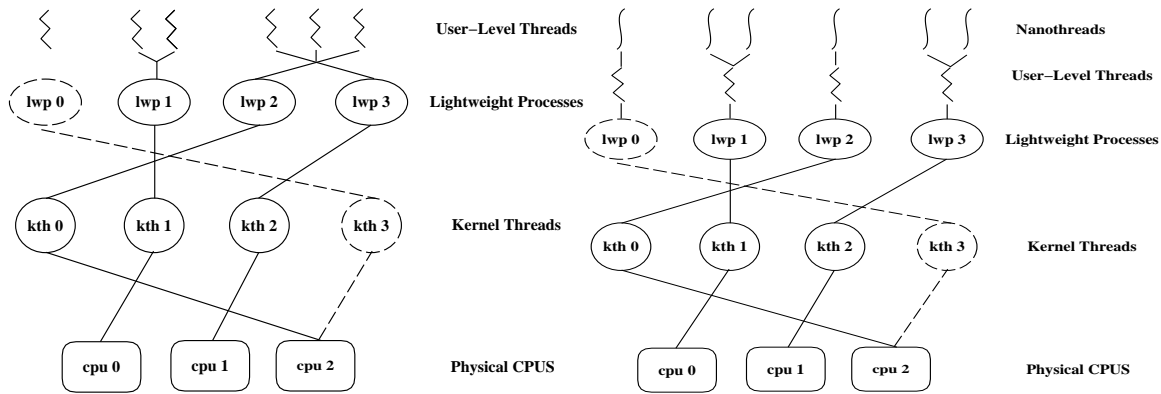


Figure 1: Solaris threads architecture and nanothreads layering. Dashed boxes indicate blocked kernel threads and LWPs.

reverts to a `nop` instruction if there are no active register windows to spill, which is the common case. This implementation led to a faster thread context switch compared to previous implementations of fast threads on the SPARC [6, 17].

The nanothreads stacks were organized similarly to the conventional organization of stacks in modern programming languages which can be optimized from the target compiler. Nesting of threads, a prerequisite for multilevel parallelization, was provided by a simple mechanism of parameter passing between threads i.e. children nanothreads access variables of their parents with pointers passed to them as regular function arguments [5].

We applied aggressive optimizations in the nanothreads runtime system to make the common case fast. We exploited the semantics of the nanothreads dependence-driven execution model [12] to provide a fast execution path for all nanothreads the register file of which need not be saved during a context switch (e.g. all slave threads in a master/slave parallel construct). At the same time, we completely eliminated threading for single and innermost parallel loops, by using work descriptors in place of nanothreads [8]. A work descriptor is a data structure which encapsulates a pointer to the body of the loop and its arguments. The nanothread that activates a loop puts the work descriptor in a specific memory location private to each processor. A processor that finds a work descriptor in the specified location, executes a wrapper function to obtain the arguments of the loop body, pushes

some of them to the stack if necessary and executes the loop body with a function call. The implementation does not require any synchronization operations and uses one-way producer-consumer communication with a single cache invalidation on each processor to schedule the parallel loop.

Work descriptors are also compliant with multilevel parallelization. Several threads may generate more than one work descriptors simultaneously and put them in distinct cache lines for execution, again without any synchronization instructions needed to execute the loops. In order to support also irregular and unbalanced parallel loops, we implemented dynamic work descriptors, i.e. work descriptors enhanced with additional loop scheduling information and synchronization constructs to support dynamic scheduling and load balancing.

Dependence-driven execution of nanothreads is based on a task-queue execution paradigm. The nanothreads runtime system uses distributed run queues to assist locality-based scheduling and a central run queue used for load balancing purposes [13]. In order to alleviate the overhead of queuing we implemented the run queues as non-blocking stacks which allow concurrent updates [15]. The run queues are coupled in a one-to-one manner with memory pools used for recycling thread contexts. The memory pools were also implemented as non-blocking stacks which take advantage of reusing cached thread contexts [10]. Spin-locks were not employed in order to avoid introducing hot spots in the memory system. A strong advan-

tage of non-blocking queues and memory pools is that the associated synchronization operations are immune to inopportune preemptions of threads from the operating system and therefore robust in the presence of multiprogramming [9].

The key feature that enables multiprogramming scalability of nanothreaded programs is a shared arena [3, 12], used to export kernel-side processor allocation decisions to the program, as well as the state of kernel threads that serve as the execution vehicles of the program. This information is used from a nanothreaded program to expose to the runtime system a degree of parallelism that matches the actual number of processors allocated to the program at any point during its execution. Based on this functionality, the user-level scheduler is able to apply intra-program control of the owned execution vehicles, to consistently assist the progress of the program along its critical path. The notable advantage of a shared arena interface is that it allows parallel programs to exchange critical scheduling information with the kernel using simply loads and stores in shared memory.

We were surprisingly fortunate to find that the native implementation of scheduler activations in the kernel of Solaris 2.6 and later versions uses a shared arena as the communication mechanism between the kernel and multithreaded programs. The kernel allocates a data structure called `sc_shared` in a page, maps this page to the user address space and pins the page to physical memory. The `sc_shared` structure can then be used to communicate the state of LWPs to the Solaris threads runtime system. Although a number of modifications to the Solaris kernel were still required to implement the interface with the nanothreads runtime system, the bulk of necessary support was already at hand and the required modifications were simple. The most important modifications needed were: (1) to communicate the actual number of physical processors allocated to the program at runtime, (2) communicate the preemption as well as the blocking/unblocking of Solaris LWPs through the scheduler activations interface, (3) let any LWP poll the state of any other LWP in the shared arena since the original implementation allows a LWP to poll only its own state and (4) modify LWP scheduling and priority control,

to implement deterministic processor yielding, i.e. let an LWP yield its processor to another specified LWP through the `sched_yield()` call.

Nanothreaded programs exploit the information in the shared arena to consistently adjust the granularity of their parallelism, in order to match the number of available physical processors. The `sc_shared` structure is used to poll the state of the LWPs allocated to a running nanothreaded program. Polling of LWP state is done to minimize LWP idling and assist the program make progress along its critical path. Each idling LWP (i.e. an LWP with no nanothread to execute in its local run queue) polls the shared arena to find out if there are other LWPs with higher intra-program priority which are not executing. LWPs with higher priority include kernel-preempted LWPs, LWPs which were previously blocked and subsequently unblocked in the kernel and previously idling LWPs, which now have a nanothread to execute in their local run queue. If any of the LWPs of that kind exists, the idling LWP blocks itself and hands-off its processor to resume the higher-priority LWP.

### 3 Performance Results

In this section we present performance results of a prototype implementation of the nanothreads runtime system in Solaris 2.6. The experiments were conducted on a Sun Enterprise 450 Server with 4 UltraSPARC-II processors clocked at 300 MHz, 2 Megabytes of secondary cache per processor and 256 Megabytes of main memory.

Figure 2 illustrates results from microbenchmarks used to evaluate the overhead of thread management in the nanothreads runtime system. The leftmost chart is extracted from a microbenchmark where each LWP forks empty nanothreads and joins for their termination repeatedly, until the program completes the execution of a million empty nanothreads. The figure is drawn in logarithmic scale and shows that the total runtime overhead of nanothreads is one order of magnitude less than the runtime overhead of user-level Solaris threads, which is in turn one order of magnitude less than the runtime overhead of Solaris

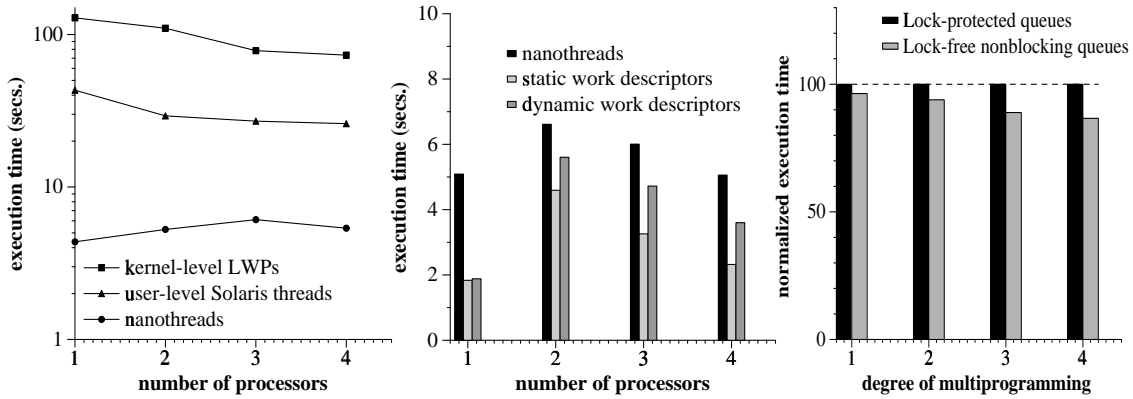


Figure 2: Evaluation with microbenchmarks. The leftmost chart is extracted from a microbenchmark where processors fork empty threads and join for their termination. The middle chart is extracted from a microbenchmark where processors execute one million parallel loops of 100 iterations each, with empty loop bodies. The rightmost chart is extracted from a microbenchmark where the processors execute one million enqueue/dequeue pairs to a shared queue.

kernel threads. The average cost to create, context-switch to, execute and synchronize for the termination of a nanothread is  $2.2 \mu s$  in the absence of contention and no more than  $3 \mu s$  with system-wide contention for the run queues and memory pools. Context-switching to and executing a nanothread cost roughly  $1 \mu s$ . Nanothread creation and queuing latencies vary between  $0.5$  and  $1 \mu s$ , depending on contention.

The middle and rightmost charts in Figure 2 illustrate the performance advantages of using work descriptors instead of threading for the innermost parallel loops and non-blocking queues instead of standard mutual exclusion with spin locks. It is important to note that work descriptors reduce the overhead of multithreading regular and irregular fine-grain parallel loops and non-blocking synchronization demonstrates a solid performance improvement over spin locks, which is magnified when the degree of multiprogramming is increased.

Figure 3 illustrates the execution time of three parallel application kernels (blocked LU decomposition, Jacobi iterative method and complex matrix multiplication with appropriately scaled problem sizes), which offer opportunities for multilevel and fine-grain parallelization. We applied multilevel parallelization for blocked LU and complex matrix multiplication, with static parallelization of the innermost loops using work descriptors. In the

Jacobi iterative method, we parallelized the innermost loops using dynamic work descriptors. We produced two versions of each benchmark using nanothreads and Solaris threads and executed the benchmarks in a dedicated system. The results favor clearly nanothreads for multilevel and fine-grain parallelization. The performance of Solaris threads can only be improved by applying more static parallelization schemes, which are however inadequate for a broad spectrum of applications.

Figure 4 shows early performance results of the nanothreads runtime system with multiprogrammed workloads. The comparison in this case is between a multiprogramming-oblivious version of the nanothreads runtime system which uses the native Solaris threads infrastructure and a multiprogramming-conscious version of the nanothreads runtime system enhanced with a preliminary implementation of the shared-arena functionality based on user-level extensions. The degree of multiprogramming corresponds to the number of instances of each application kernel launched simultaneously during an experiment. Each instance requests initially 4 processors to run on, therefore a constant degree of multiprogramming is guaranteed in each case. Despite the clear advantages of the multiprogramming-conscious mechanisms of the nanothreads runtime system, there is plenty of room for improvements. We are still working on the integration of the nanothreads kernel interface

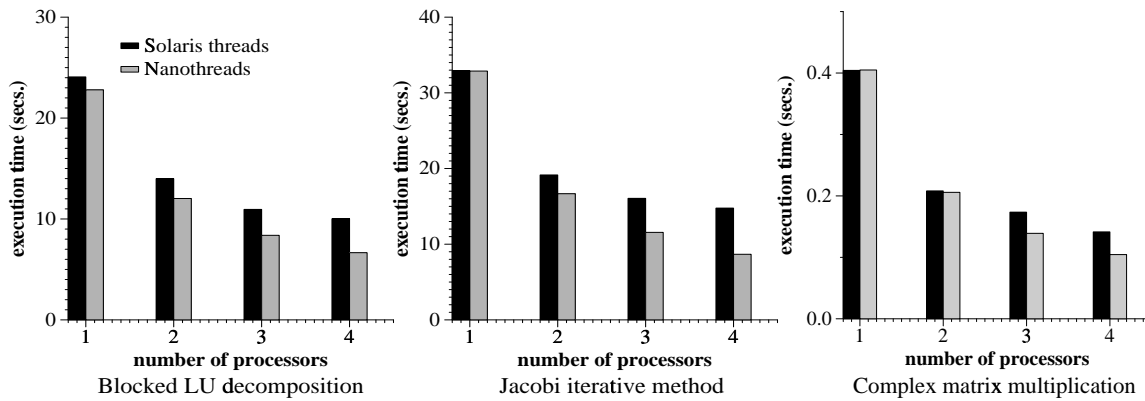


Figure 3: Application kernels performance.

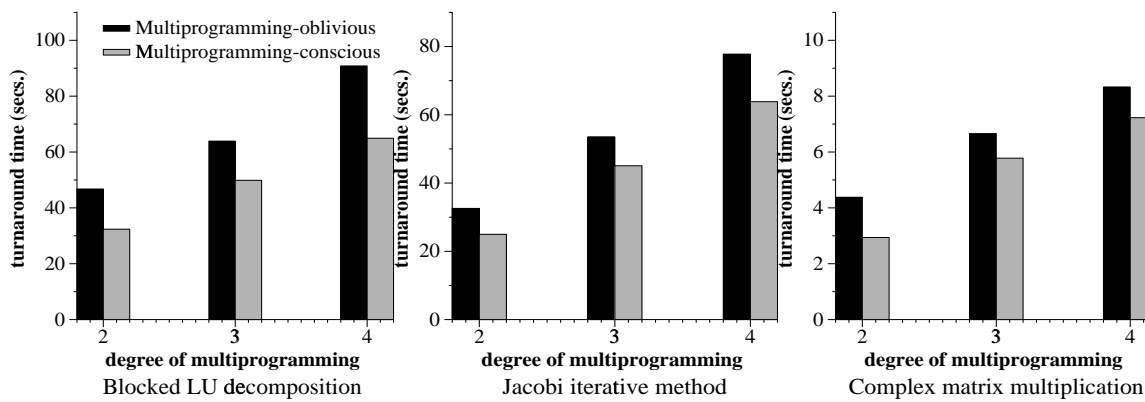


Figure 4: Multiprogrammed workloads performance.

in Solaris, focusing mainly on the integration of the interface with dynamic space sharing schemes in the Solaris kernel scheduler.

## 4 Conclusions

We presented the implementation details of a nanothreads runtime system for multiprocessors running the Solaris operating system. The goal of the implementation was to provide fine-grain multilevel parallelization and multiprogramming scalability at affordable runtime costs. We applied aggressive optimizations to reduce multithreading overhead and make the common case fast. The design of the nanothreads runtime system is not orthogonal to Solaris threads. Nanothreads augment the Solaris threads architecture and exploit the underlying infrastructure to the maximum extent possible. Solaris LWPs serve as the execution vehicles for nanothreads and existing kernel

data structures are used effectively to implement multiprogramming-conscious mechanisms. The overall threads architecture provides an environment for efficient parallelization and execution of applications on multiprogrammed shared-memory multiprocessors based on the SPARC architecture.

## Acknowledgements

We are grateful to Constantine Polychronopoulos and our partners in the NANOS project. This work is funded by the European Commission under the ESPRIT IV Project No. 21907 (NANOS).

## References

- [1] T. Anderson, B. Bershad, E. Lazowska and H. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management*

- of *Parallelism*. ACM Transactions on Computer Systems, 10(1), pp. 53–79, 1992.
- [2] N. Arora, R. Blumofe and C. Plaxton. *Thread Scheduling for Multiprogrammed Multiprocessors*. Proc. of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, Puerto Vallarta (Mexico), 1998.
- [3] D. Craig. *An Integrated Kernel-Level and User-Level Paradigm for Efficient Multiprogramming*. MSc Thesis, University of Illinois at Urbana-Champaign, 1998.
- [4] M. Frigo, C. Leiserson, and K. Randall. *The Implementation of the Cilk-5 Multithreaded Language*. Proc. of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation, pp. 212–223, Montreal (Canada), 1998.
- [5] M. Girkar, M. Haghghat, P. Grey, H. Saito, N. Stavrakos and C. Polychronopoulos. *Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems*. Intel Technology Journal, Q1 Issue, 1998.
- [6] D. Keppel. *Tools and Techniques for Building Fast Portable Thread Packages*. Technical Report UW-CSE-93-05-06, University of Washington at Seattle, 1993.
- [7] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé. *A Library Implementation of the Nano-Threads Programming Model*. Proc. of the Second EuroPar Conference, pp. 644–649, Lyon (France), 1996.
- [8] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta. *Threads Fork/Join Techniques for Multilevel Parallelism Exploitation in NUMA Multiprocessors*. Proc. of the 13th ACM International Conference on Supercomputing, Rhodes (Greece), 1999.
- [9] M. Michael and M. Scott. *Nonblocking Algorithms and Preemption-Safe Locking in Multiprogrammed Shared Memory Multiprocessors*. Journal of Parallel and Distributed Computing, 54(2), pp. 162–182, 1998.
- [10] D. Nikolopoulos, E. Polychronopoulos and T. Papatheodorou. *Efficient Runtime Thread Management for the Nano-Threads Programming Model*. Proc. of the Second IPPS/SPDP Workshop on Runtime Systems for Parallel Programming, LNCS Vol. 1388, pp. 183–194, Orlando (USA), 1998.
- [11] C. Polychronopoulos. *Multiprocessing vs. Multiprogramming*. Proc. of the 1989 International Conference on Parallel Processing, pp. II-223–II-230, St. Charles (USA), 1989.
- [12] C. Polychronopoulos, N. Bitar and S. Kleiman. *Nan threads: A User-Level Threads Architecture*. CSRD Technical Report No. 1297, University of Illinois at Urbana-Champaign, 1993.
- [13] E. Polychronopoulos and T. Papatheodorou. *Scheduling User-Level Threads on Distributed Shared Memory Multiprocessors*. Technical Report 010498, HPIS Laboratory, University of Patras, 1998.
- [14] E. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. Papatheodorou and N. Navarro. *Kernel-Level Scheduling for the Nano-Threads Programming Model*. Proc. of the 12th ACM International Conference on Supercomputing, pp. 337–344, Melbourne (Australia), 1998.
- [15] J. Valois. *Lock-Free Data Structures*. PhD Dissertation, Rensselaer Polytechnic Institute, 1995.
- [16] D. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, 1994.
- [17] B. Weissman, B. Gomes and J. Feldman. *Active Threads: Enabling Fine-Grained Parallelism in Object-Oriented Languages*. Proc. of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (USA), 1998.