Tying Memory Management to Parallel Programming Models

Ioannis E. Venetis* and Theodore S. Papatheodorou

High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras, Rion 26500, Greece
http://www.hpclab.ceid.upatras.gr

Abstract. Stand-alone threading libraries lack sophisticated memory management techniques. In this paper, we present a methodology that allows threading libraries that implement non-preemptive parallel programming models to reduce their memory requirements, based on the properties of those models. We applied the methodology to NthLib, which is an implementation of the Nano-Threads programming model, and evaluated it on an Intel based multiprocessor system with Hyper-Threading and on the SMTSIM simulator. Our results indicate that not only memory requirements drop drastically, but that execution time also improves, compared to the original implementation. This allows more fine-grained, but also larger numbers of parallel tasks to be created.

1 Introduction

Efficiency of parallel programming models has traditionally been measured in terms of two important metrics. On the one hand, execution time is used to indicate whether threading libraries and parallel applications have been implemented effectively. On the other hand, the amount of resources that are required to express and execute parallel tasks has also been of great importance. Especially usage of memory, which might be a scarce resource on some parallel systems, has been carefully analyzed in several cases. Work conducted towards this direction, mainly targets threading libraries that support multithreaded languages, i.e., parallel languages that support dynamic thread creation. This is due to the fact that the accompanying compilers are able to perform powerful analysis of memory requirements per function and propagate this information to the library.

In contrast, stand-alone threading libraries lack the knowledge about the memory requirements of an application and must therefore be pessimistic about them. Due to this fact, two different paths to implement such libraries have emerged. The basic requirement of the first one is to reduce execution time at all costs. Libraries implemented under this scheme, which we will refer to as

* This work has been carried out while the first author was supported by a grant from the 'Alexander S. Onassis' Public Benefit Foundation and the European Commission through the 'POP' IST project (Grant No.: IST-2001-33071).

Descriptor on Stack (DOS), usually allocate a large region of memory during thread creation. This region is logically divided into two parts, one describing the parallel task (Descriptor) and the other being the stack of the task during execution. Although common sense suggests that this is a fast method to create threads, memory requirements are often excessive. Taking into account that contemporary stand-alone threading libraries set the default stack size somewhere between 1 and 4MB, makes it obvious that those libraries cannot support large numbers of threads. In order to overcome this problem, supporters of the second implementation strategy suggest that the first priority should be to minimize memory requirements, even if execution time suffers. In this case, during thread creation time, only a small descriptor is allocated and initialized, whereas a larger stack is assigned to the task when it is selected to run. We will refer to this strategy as *Lazy Stack Allocation (LSA)*. Effectively, stacks are traded in favour of smaller descriptors.

Current developments in computer architecture, such as *Simultaneous MultiThreading (SMT)* [8], *HyperThreading* [3] and multicore processors, allow efficient execution of more fine-grained parallelism, in addition to a larger number of parallel tasks. This allows applications to express more of their inherent parallelism, creating a number of threads that might exceed the number of available execution contexts (ECs). Our view is to create an infrastructure that will allow stand-alone threading libraries to efficiently support the above execution scheme. In addition to these observations, taking into account that the above architectures usually have a lower memory per EC ratio, leads us to the conclusion that the LSA implementation strategy is more appropriate for such systems. However, those libraries are usually slower, thus invalidating the means provided by modern processors to efficiently execute parallel tasks. Hence, it becomes obvious that a new approach to tackle this problem is necessary, which will combine the benefits of both approaches.

In this paper we present a methodology that allows stand-alone threading libraries that implement non-preemptive parallel programming models to reduce their memory requirements. This is accomplished by taking into consideration the properties of this specific parallel programming paradigm. A prerequisite, however, is that threading libraries should be LSA enabled. Firstly, we present a method to convert a DOS into a LSA enabled library. In addition, this method takes into account an important factor that greatly affects speed of DOS enabled libraries, i.e., their self-identification mechanism. Based on this, we take a step further, compared to previous approaches, introducing two methods that reduce memory requirements even more. The first one allows us to compute *a priori* the total number of stacks that are required to run an application. The second one improves on LSA, by directly handing the stack of a terminating thread to the next one that should run on a processor. We will refer to this method as *Direct Stack Reuse (DSR)*. We demonstrate for the first time, to the best of our knowledge, that LSA and DSR enabled libraries can actually outperform DOS enabled libraries, in terms of both, execution time and memory requirements.

The rest of this paper is organized as follows: Section 2 presents related work, with respect to memory management techniques under several parallel program-

ming models. Section 3 presents how to apply our methodology to convert DOS into LSA enabled libraries. In Section 4 and Section 5 we present how to further reduce memory requirements. In Section 6 we experimentally evaluate our approach. Finally, in Section 7 we conclude our paper.

2 Related Work

As already mentioned, much work has been done to reduce memory requirements in threading libraries that support multithreaded languages. For example, in the Lazy Task Creation [5] model, a thread is implemented as a serial call to a function, which allows it to run in the stack of the parent. If, however, the child suspends execution or more parallelism is required, the recorded return address of the parent is assigned to another processor and the corresponding stack frames are copied, in order for the parent to continue execution. The Lazy Threads [2] model employs several representations of parallelism and makes the compiler responsible for selecting the most efficient in each case. Accordingly, the compiler decides which is the best representation of a stack, after statically analyzing each function. For serial execution, a conventional stack is used, for threads with small and medium sized data a structure know as a *stacklet* is used, whereas for larger data sets a separate memory region is allocated. The usual case is to use a stacklet, which is a memory region that can hold more stack frames. However, initialization and release of a stacklet are quite expensive operations. The Capriccio [10] threading library also uses data from static analysis of the compiler. Similarly to the previous model, more stack frames are put in each memory region, with the associated management cost. A new region is allocated at the check points that the compiler inserts, according to the performed analysis. However, Capriccio implements an 1:N model, where more user-level threads are executed on top of only one kernel-level entity. Hence, true parallelism cannot be exploited and some of the optimizations are not valid in M:N models.

An interesting approach, that tries to simplify development of compilers for multithreaded languages, is the one proposed in StackThreads [7]. It provides basic functionality to compilers, in order to map the execution model of multithreaded languages to the execution model of the C programming language. More advanced management of stack frames can be built on top of this functionality. Memory is managed in the library, through the information that the compiler has to pass to it. Although generality is an important concern in StackThreads, that work targets a different set of threading libraries than our work.

With respect to stand-alone threading libraries, TiNy Threads [1] targets the Cyclops64 system, which has extremely limited memory. Only 4800KB are available for 150 ECs. It uses the DOS model and due to increased memory requirements has to limit the number of threads that can be created, actually invalidating the objective of the architecture. In threading libraries that are preemptive, such as POSIX threads, threads are usually created and immediately put into a ready queue for execution. In these cases, more threads than there are processors are usually active. This, in turn, implies that a large number of

stacks must be available, in order to keep the state of threads that have run but are currently preempted. As a consequence, separation of descriptors and stacks can only be applied to efficiently recycle objects in such cases and not reduce usage of memory. In this paper, however, we target non-preemptive threading libraries, which have well defined entry and exit points for a thread. Exploiting this property, is what differentiates our work from previous approaches.

3 Retaining a Fast Self-Identification Mechanism

DOS enabled libraries are thought to be fast for two reasons. Firstly, no stack has to be assigned to each thread before execution, because it has already been allocated at thread creation time. Secondly, the allocated memory region, that is split between the descriptor and the stack, is usually aligned at the region's size. This allows a thread to quickly perform self-identification, i.e., find the starting address of it's own descriptor and acquire important information about it's status. For example, if a 1MB($=2^{20}$ bytes) memory region is allocated, the starting address should have it's 20 last bits zero. Self-identification is performed in this case by reading the current value of the Stack Pointer and clearing the last 20 bits. The result is always the starting address of the memory region. Adding the size of the memory region and subtracting the size of the descriptor, returns the starting address of the latter. By dereferencing this value, all the information contained in the descriptor can be obtained. For a more detailed description of the mechanism, including figures, we refer the reader to [9].

Our first requirement, while switching to a LSA model, is to retain a fast self-identification mechanism. In order to achieve this goal, our methodology requires us to follow two steps. Firstly, stacks should be aligned as in the DOS model. We must point out that the stack under LSA is actually the same as the memory region in the DOS model, whereas descriptors are allocated separately. Secondly, after a thread has been selected to run and a stack has been assigned to it, a pointer to the descriptor (instead of the descriptor itself) should be put at the top of the stack. Thus, self-identification is performed almost in the same manner as in DOS enabled libraries. The difference lies in the last step, where the size of a pointer is subtracted from the computed value, instead of the size of a descriptor. This returns a pointer to the descriptor, which can be dereferenced, as in the DOS case. Hence, this mechanism is as fast as the original one.

4 Direct Stack Reuse

DOS requires a large memory region for each thread. If more threads than processors are created, this leads to unnecessarily high memory consumption. In contrast, LSA, in combination with the fact that a non-preemptive model guarantees that a thread will not be interrupted, allows the stack of a terminating thread to be inserted into a recycling queue and another stack to be assigned to the next thread. The same process is repeated for every thread that terminates,

on each processor. Hence, LSA requires only two stacks per processor. If, however, only one thread is created for each processor, LSA will use the recycled stack when a new thread finally arrives, thus the number of stacks will be equal to the DOS case, in addition to a number of small descriptors.

LSA is a widely used method to reduce memory usage. However, with non-preemptive models more improvements can be achieved. Our second requirement, when switching to a LSA model, is to reduce the time required to assign a stack to a thread that is ready to run. When a context-switch occurs, under a non-preemptive model, a thread is actually terminating and it's stack is not needed anymore. Due to this observation, it is obvious that the stack of that thread can be directly reused by the new thread, without accessing queues or allocating a new one. This reduces time to find a stack for the new thread. Setting up the stack in this case, is as expensive as in the DOS case. The difference is that initialization just happens at a different point in the execution path.

Two important points have to be made clear, the first being that DSR can only be applied if LSA is also active. The second point is that DSR is a complementary mechanism to the recycling queues. Someone could conclude that recycling queues could be dropped from a library, since each thread directly uses the stack of the previous thread. However, there are cases where recycling queues are necessary. For example, a thread might block and voluntarily release the processor it is running on. In this case, the user-level scheduler selects a new thread for that processor. Obviously, the stack of the thread that blocked must be preserved, in order for it to be able to resume execution. Hence, the newly selected thread needs a new stack, which it will request from the queues.

5 Calculating the Number of Required Stacks

Although LSA already contributes to reduced memory requirements and DSR improves on that, eliminating one stack per processor for switching to a new thread, the non-preemptive nature of the programming models that are considered allow us to go even further. Specifically, it is possible to calculate a priori the number of required stacks that are necessary to run an application. To demonstrate this, we will use as an example NthLib [4], a threading library that implements the Nano-Threads programming model [6]. However, this specialization has an effect only in the initialization phase of a library. During context-switches, the following reasoning applies to every threading library.

Suppose that initially only LSA is enabled and that an application requests P processors. In this case, the library must create $P-1$ Virtual Processors (VPs) more, since the first VP is the one that started the application. Currently, the requirements for the initialization phase are one stack for the first VP and two for each other, giving a total of $2 \cdot P - 1$. If DSR is also enabled, then there is no need for additional stacks. If not, then as soon as a VP, except of the first one, receives the first user-level thread for execution, one of the stacks is recycled, as it is thought to be the stack of the previous thread. Therefore, only one more stack is required per VP to perform context-switches, which sums up

to $(2 \cdot P - 1) + (P - 1) = 3 \cdot P - 2$. Finally, one more stack is required for the main thread, because it voluntarily blocks and joins the other VPs in the parallel phase. Therefore, the final sums are $3 \cdot P - 1$ for LSA and $2 \cdot P$ for DSR.

This information can be used by threading libraries. During initialization, allocation of all stacks can be performed at once. In order to enforce the memory alignment requirements of the library, some additional pages of memory must be allocated. However, after the first stack has been correctly aligned, all subsequent stacks in that memory region will also be aligned. This is in contrast to DOS, where additional memory has to be allocated for each new thread, which is wasted. If we were to free that memory, execution time would suffer. Since the application is still in a serial phase, stacks can be put into the corresponding recycling queues without using any locking mechanism. This has two advantages. Firstly, the library requests only once memory for stacks. Memory allocators are usually slow when large memory areas are requested. Under DOS, this cost is paid for every thread. Under LSA and DSR, the cost is amortized among all threads that will run on the allocated stacks. Secondly, each processor is assigned the total number of stacks it requires during execution. This reduces contention on the recycling queues of the stacks to the minimum. Finally, it is a priori known that all allocated stacks will be used and no memory will be wasted.

Although the number of stacks can be predicted, the number of descriptors cannot. Despite that fact, we propose a similar pre-allocation technique for them. The first time a descriptor is requested, a larger area of memory is allocated. However, in contrast to stacks, all descriptors that fit into this area are not directly put into queues, due to the fact that mutual exclusion would be necessary. In addition, only the main thread of an application usually creates threads, hence spreading descriptors among all recycling queues would be inefficient. Therefore, each time a descriptor is needed, and none can be found in a recycling queue, we atomically increase the base address of the allocated area by the size of a descriptor and return the previous address. Since contention is very low, due to the fact that usually only one thread creates others, this atomic operation is very likely to complete very fast. Only when the allocated area is exhausted, does the library request more memory for descriptors. Currently, $2 \cdot P$ descriptors can fit into the area that is each time allocated in the new implementation of NthLib. In combination with the fact that a descriptor is only 512 bytes large, one can conclude that this is an effective method to reduce the number of expensive memory allocation requests, without actually sacrificing memory.

6 Experimental Evaluation

In order to evaluate the efficiency of our methodology, we applied it to NthLib. The original implementation of NthLib is DOS based. The new implementation has been developed so as to support all designs that were described, i.e., DOS, LSA and DSR. The one that is each time used is defined during compilation of the library. Supporting all designs was intentional, in order to make comparison among them easier. Having only one library, makes our results independent of other differences and details that two separate implementations would have.

Table 1. Hardware configuration of the experimentation platform.

	Intel processor based system	SMTSIM
Processors	4 Intel Xeon MP HTs, 2 GHz, 2 execution contexts/processor	1 Alpha based, 8 execution contexts
L1 Data Cache	8KB shared, 4-way assoc.	32KB, 2-way assoc., 10-cycle miss latency
L1 Inst. Cache	12KB shared execution trace	32KB, 2-way assoc., 10-cycle miss latency
L2 Cache	512KB shared, unified, 8-way assoc.	256KB, 2-way assoc., 15-cycle miss latency
L3 Cache	1MB shared, unified, 8-way assoc.	2MB, 2-way assoc., 125-cycle miss latency
D-TLB	64 entries	128 entries
I-TLB	2x64 entries	48 entries
DRAM	2GB	Depends on host system

The first system we used to evaluate our approach is a 4-processor, Hyper-Threading enabled system, running Linux 2.6.8. The second one is SMTSIM [8], a simulator that implements an Alpha processor with 8 ECs. More detailed hardware characteristics for both systems are summarized in Table 1. The compiler used is gcc 4.0.2 for both platforms, at the highest optimization level (-O3).

Due to space limitations, we present results for only one benchmark. We refer the reader to [9], for a more detailed description of applying our methodology to NthLib and a more thorough evaluation. The benchmark that we used, which we will refer to as *Empty*, follows the fork/join model. The master thread creates one million empty nano-threads, whereas the slave processors dispatch and execute them. The master thread blocks after it has created all threads, hence calling the user-level scheduler and joining the other processors to execute threads. This benchmark is appropriate for estimating the pure run-time overhead of thread management in NthLib. Moreover, it can be used to determine the number of stacks that an application requires and to estimate the minimum number of descriptors that must be allocated. This is due to the fact that nano-threads perform no computation in this benchmark. Therefore, they are consumed as fast as possible by the slave processors and are immediately recycled.

Fig. 1 summarizes the results for this benchmark. They are normalized with the time of the slowest benchmark, which is when LSA is enabled and the benchmark is run on one EC. The absolute execution times in this case were 2,56 seconds for the Intel based system and 643,5 million simulated clock cycles for SMTSIM. The stack size used for each nano-thread was set in all runs to the quite small size of 32KB, in order to allow the benchmark to successfully complete in most cases under the DOS scheme. If either LSA or DSR is enabled, pre-allocation of stacks and descriptors is also enabled. For SMTSIM, the horizontal axis represents the number of ECs used. For the Intel based system, the numbers of physical processors and ECs used on each one of them are mentioned. For example, (4, 1) means that 1 EC was used on each one of the 4 physical pro-

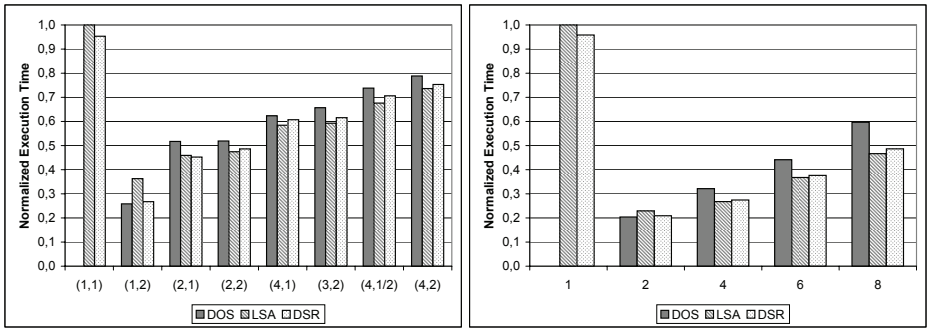

Fig. 1. Normalized execution time for *Empty* on the Intel based system and SMTSIM.

cessors. A special case is the one denoted with (4, 1/2), where 2 ECs were used on 2 physical processors and 1 EC on the other 2 physical processors. Finally, notice that the benchmark could not complete when DOS was enabled and it was run on one processor, due to excessive memory requirements. The master thread must first create all threads in this case, before blocking to execute them.

The results indicate that LSA and DSR are from 2,65% (DSR, (4,1)) up to 12,54% (DSR, (2,1)) faster than DOS, on the Intel platform. For SMTSIM, the range is between 14,66% (DSR, 4) and 21,82% (LSA, 8). The exception occurs when two ECs are used on only one processor. In this case, DSR is 3,49% and LSA 40,41% slower than DOS, for the Intel platform, whereas for SMTSIM the difference is 2,48% and 12,38% respectively. More detailed measurements revealed that the cause of this inefficiency is the contention on the recycling queues of both, the stacks and the descriptors. Under LSA and DSR, nanothreads are created and also start executing faster after they have been selected to run. This means that both ECs try to acquire access to the queues in smaller time intervals. In combination with the fact that resources of the processor are shared between the ECs and one of them may stall, if a resource is not available, explains this odd behavior. DSR is faster than LSA in this case, due to the fact that stacks are not recycled but directly reused. This alleviates the queueing system significantly. Moreover, the differences for SMTSIM are smaller, compared to the Intel platform. Additionally, we observe that execution time rises, as more ECs are used, although with a smaller pace in the case of SMTSIM. The fact that SMTSIM exhibits better behaviour in all cases can be attributed to the following facts. Firstly, SMTSIM simulates 8 ECs on one processor. The Intel platform, in contrast, is a SMP system, where communication costs among processors are quite higher. Secondly, resources of the simulated processor in SMTSIM are dynamically shared among ECs. In the second system, however, resources of a processor are statically divided between both ECs. Therefore, even if a resource is available on one of them, the other cannot take advantage of it. Lastly, SMTSIM uses a very efficient hardware implementation of locks, based on the concept of a *lockbox* [8], which is exploited in our library and significantly reduces synchronization time among threads.

Table 2. The number of required Stacks (S) and Descriptors (D) for *Empty*.

	Intel processor based system				
	DOS	LSA		DSR	
	S	S	D	S	D
(1,1)	-	3	1000003	2	1000003
(1,2)	13012	6	284877	4	204662
(2,1)	9602	6	176701	4	107078
(2,2)	7342	10	104896	8	76378
(4,1)	10909	10	80893	8	73132
(3,2)	9993	16	73760	12	86505
(4,1/2)	8558	16	70684	12	76668
(4,2)	10496	22	65270	16	73812

	SMTSIM				
	DOS	LSA		DSR	
	S	S	D	S	D
1	-	3	1000003	2	1000003
2	8	5	31	3	15
4	9	7	14	7	14
6	11	11	16	11	15
8	13	13	17	12	16

The other important factor that our implementation tries to minimize, apart from execution time, is usage of memory. Table 2 summarizes the memory requirements of our benchmark. Starting with the results for the Intel platform, it becomes obvious that savings in memory are significant. The biggest difference between DOS and LSA appears when both ECs are used on each one of the physical processors and is 90,07%. The biggest difference between DOS and DSR appears when one EC is used on all physical processors and is 89,45%. The smallest difference appears in both cases when one EC is used on each of two physical processors and is 65,75% for LSA and 75,39% for DSR.

Different results are acquired for SMTSIM, where memory requirements for all cases are almost identical. This difference, compared to the Intel platform, can be explained, if we take into consideration that SMTSIM does not run an OS and delays that origin from it are not accounted for. As an example of the importance of this fact, we mention that on the Intel platform, a thread is created in about 30000 clock cycles, under the DOS scheme, whereas the time required to insert it into a ready-queue is only about 250. Almost all of the time to create the thread is spent in the OS, in order to allocate the required memory. In SMTSIM, however, this time is not measured and a thread is created in 60 cycles and inserted into a queue in 100. Therefore, we believe that for fine-grained benchmarks, that frequently interact with the OS, SMTSIM is not as accurate as required. Consequently, we believe that the results obtained on the Intel platform, with respect to memory requirements, reflect better reality. Furthermore, we believe that differences in execution time between DOS and both, LSA and DSR, would be higher for SMTSIM, if the time for memory allocation had been taken into account. However, SMTSIM gives a good estimation of execution times in all cases and can be used more reliably for applications where each thread has to perform more computations [9].

7 Conclusions

In this paper we presented a methodology that can be applied to non-preemptive parallel programming models, in order to reduce their memory requirements. We

used a widely known methodology to convert a DOS into a LSA enabled library, demonstrating that it is possible to retain a fast self-identification mechanism. Furthermore, taking into account the fact that most contemporary high performance threading libraries implement non-preemptive parallel programming models, we introduced two more methods to reduce memory requirements. Those are based on the properties of non-preemptive programming models, which is what differentiates our work from previous approaches. Specifically, we introduced a mechanism that allows the stack of a terminating thread to be directly reused by the thread that is next to be run. In addition, we demonstrated how it is possible to calculate a priori the total number of stacks that an application requires. The latter can be exploited to reduce the amount of memory that would otherwise be wasted, due to the alignment requirements of memory regions and stacks. Finally, our performance evaluation proved that combining all of the above techniques, not only drastically reduces memory requirements to represent parallelism in threading libraries, but that it can also be faster than the traditional DOS approach, in contrast to general belief.

References

1. J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Proceedings of the 5th Workshop on Massively Parallel Processing*, Denver, Colorado, April 2005.
2. S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, Volume 37, Issue 1:5–20, August 1996.
3. D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, Volume 6, Issue 1:4–15, February 2002.
4. X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A Library Implementation of the Nano-Threads Programming Model. In *Proceedings of the 2nd International EuroPar Conference*, pages 644–649, Lyon, France, August 1996.
5. E. Mohr, D. A. Kranz, and Jr. R. H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, Volume 2, Issue 3:264–280, July 1991.
6. C. Polychronopoulos, N. Bitar, and S. Kleiman. Nanothreads: A User-Level Threads Architecture. Technical Report 1297, CSRD, University of Illinois at Urbana-Champaign, 1993.
7. K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/MP : Integrating Futures into Calling Standards. Technical Report TR 99-01, University of Tokyo, 1999.
8. D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, S. Margherita Ligure, Italy, 1995.
9. I. E. Venetis and T. S. Papatheodorou. A Time and Memory Efficient Implementation of the Nano-Threads Programming Model. Technical Report HPCLAB-TR-210106, High Performance Information Systems Laboratory, January 2006.
10. R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 268–281, Bolton Landing, New York, October 2003.