

A Transparent Operating System Infrastructure for Embedding Adaptability to Thread-Based Programming Models

Ioannis E. Venetis¹, Dimitrios S. Nikolopoulos^{1,2*}, and Theodore S. Papatheodorou¹

- ¹ High Performance Information Systems Laboratory, Department of Computer Engineering and Informatics
University of Patras, Rion 26500, Greece
<http://www.hpclab.ceid.upatras.gr>
- ² Computer and Systems Research Laboratory, Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL, 61801
<http://www.csrd.uiuc.edu>

Abstract. Parallel programs executing on non-dedicated SMPs should be *adaptive*, that is, they should be able to execute on a dynamically varying environment without loss of efficiency. This paper defines a unified set of services, implemented at the operating system level, which can be used to embed adaptability in any thread-based programming paradigm. These services meet simultaneously three goals: they are highly efficient; they are orthogonal and transparent to the multithreading programming model and its API; and they are non-intrusive, that is, they do not compromise the native operating system's resource management policies. The paper presents an implementation and evaluation of these services in the Linux kernel, using two popular multithreading programming systems, namely OpenMP and Cilk. Our experiments show that using these services in a multiprogrammed SMP yields a throughput improvement of up to 41.2%.

1 Introduction

The advent of shared-memory multiprocessors (SMPs) and the broad spectrum of applications in which SMPs are employed have popularized thread-based parallel programming models. The multithreading paradigm is a convenient means for implementing concurrent tasks that communicate through a virtually or physically shared memory address space. A multitude of multithreading interfaces for parallel programming is in use, including standardized interfaces like POSIX threads [4] and experimental systems that serve as the backend of specific algorithmic (e.g. Cilk [2]) or compilation (e.g. Nanthreads [11]) frameworks.

* This work has been carried out while the second author was with the High Performance Information Systems Laboratory, University of Patras, Greece.

The metrics used so far as evaluation criteria of multithreading systems are primarily the overhead of thread management and the quality of the user-level thread scheduler in terms of response time. Recently, it has been recognized that the performance of applications and the overall throughput of the system in the presence of *multiprogramming* are becoming factors of increasing importance for the evaluation of multithreading systems. This stems from a shift from a dedicated to a non-dedicated mode of use of modern SMPs. In a non-dedicated mode of use, an SMP is shared among multiple programs and the workloads submitted to it often exceed the system's capacity, thus necessitating some form of resource multiplexing by the operating system.

From the perspective of a multithreading system, coping with multiprogramming and resource sharing requires a degree of *adaptability* embedded in the system. Each multithreading program should be capable of executing efficiently on a dynamically varying number of processors. The program should be able to both resume preempted computation, should it lose a processor and utilize a newly granted idle processor.

The literature provides a wealth of solutions for attacking the performance bottlenecks that arise from the interference between multiprocessing and multiprogramming [1,3,6]. However, little effort has been spent on the transparent integration of these solutions and multithreading programming models in a generic, model-independent manner. Existing frameworks for efficient multiprogrammed execution either pose stringent requirements on the multithreading model, or depend on the semantics of the multithreading model.

Some representative examples depict the situation. Process control and relevant frameworks [14] require that the multithreading system uses a task-queue execution paradigm. Hood [10], an extension of Cilk that encompasses a non-blocking user-level scheduler, works only under the assumption that the task queues are organized as stacks with complicated semantics and that the computation is expressed in a strict fork/join style, analogous to that of divide-and-conquer algorithms. The Nanothreads [11] architecture requires a compiler that parallelizes programs in multiple levels of task granularity and injects code to select the thread granularity that maximizes efficiency at runtime. It is rather unfortunate that despite the wealth of solutions, popular thread-based programming standards like POSIX threads and higher-level programming models based on thread-based systems, such as OpenMP [9], lack the required adaptability to ensure efficient parallel execution in the presence of multiprogramming.

This paper addresses the problem of embedding multiprogramming adaptability to thread-based systems in a totally transparent manner, that is, independently of the internals of the thread-system and without modifying its API. The ultimate purpose of our work is to improve the scalability of any thread-based system on a multiprogrammed SMP, regardless of the theoretical thread model that the system implements, or its implementation idiosyncrasies.

We present the design of a kernel-level infrastructure, which provides a minimal set of services for multiprogramming adaptability. These services can be used by any threads library and include a bidirectional communication channel

between the operating system and multithreaded applications, a second-level scheduler that controls the execution of adaptive multithreaded applications and two mechanisms that enable applications to eliminate idling at synchronization points. These mechanisms are built on top of existing operating system components, however they do not alter the available resource management policies of the operating system. Furthermore, the services do not require changes in the source code or recompilation of multithreaded applications.

We implemented our services in the Linux kernel and modified the LinuxThreads library [7], which is an implementation of the POSIX 1003.1c threads standard for Linux-based systems. We used these modified versions to embed adaptability in two multithreading programming models with radically different characteristics, namely OpenMP and Cilk. Our results from multiprogrammed executions of programs written with both programming models show that our services achieve sizeable improvements in throughput (from 4.6% up to 41.2%) compared to the throughput of the native Linux kernel.

The remaining of this paper is organized as follows: Section 2 gives a general overview and Section 3 describes a prototype implementation of the proposed infrastructure in the kernel of Linux. Section 4 describes the additions required in threads libraries to use the kernel-level infrastructure and focuses on the modifications made in the LinuxThreads library. Section 5 provides experimental evidence on the value of our approach and Section 6 concludes the paper.

2 The Proposed System Software Architecture

The goal of the proposed system software architecture is to facilitate execution adaptability under multiprogramming. This implies that applications must be armed with mechanisms that enable them to adapt to a dynamically changing execution environment. The most valuable among these mechanisms are those that eliminate idling at synchronization points and minimize cache and memory interference by space sharing the processors of the system. It has been shown that adapting the number of threads of an application to the available number of processors and resuming threads that have been preempted while executing useful work in the critical path can improve significantly the performance of the operating system scheduler, in terms of execution time and throughput [8]. These services are orthogonal to both the programming model and the multithreading back-end used to express parallelism. As a consequence, the natural levels of implementation and exploitation of such services are the operating system and the run-time threads libraries respectively.

An implementation of these services should ideally satisfy three conflicting requirements. The first is efficiency; the overhead of the services must be kept at a minimum. The second is transparency. These services should be injected to any thread-based programming model or library, without having to change the native API. Non-intrusiveness, finally, implies that the services do not alter the fundamental resource management policies of the operating system, which

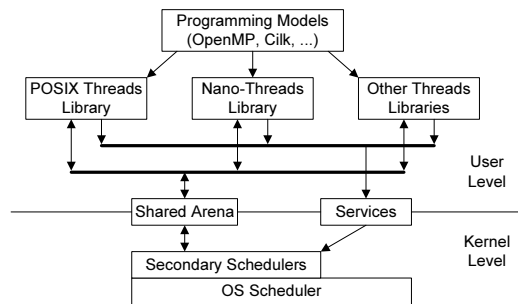


Fig. 1. The Proposed System Software Architecture

are always designed to satisfy a broader spectrum of criteria, rather than just executing efficiently multithreaded programs.

A schematic diagram of a kernel-level infrastructure designed to fulfill these requirements is depicted in Figure 1. There are three components that interact to provide the desired functionality. The first is a bidirectional communication channel between the operating system and each application. This channel allows applications to inform the kernel about their needs on resources and be informed about the scheduling decisions taken by the kernel. Since scheduling decisions are taken frequently, the communication channel has to be asynchronous and very efficient, in order to let applications access scheduling information with low overhead and only when required. A *shared arena* [11], i.e. a set of memory-resident pages mapped to both the user and the kernel address spaces can serve this purpose. Using the shared arena, applications exchange information with the kernel through loads and stores in shared memory.

The second component of our infrastructure is a non-intrusive scheduler, built on top of the native kernel scheduler. This scheduler controls only adaptive multithreading applications and its objectives are to distribute fairly CPU time among adaptive applications and apply suitable scheduling policies that enable applications to execute efficiently on a dynamically varying set of resources.

The last component of our infrastructure encompasses a set of services that let the applications effectively utilize idle processors. Most threads libraries overcome both the problem of eliminating idling at synchronization points and the implications introduced by inopportune preemptions of threads in an essentially similar way, i.e. by having idle threads yield their processors to other potentially non-idle threads. Exploiting this similarity makes it possible to provide a limited set of common services that can be used by most existing threads libraries.

Implementing such services in the operating system and exploiting them in threads libraries allows user applications to take advantage of these services in a totally transparent manner. There is no need to modify or recompile an application to use these services, provided that it is dynamically linked with a threads library. A new distribution of the library that uses the kernel-level infrastructure is sufficient to arm these applications with the desired adaptability.

3 Kernel Interface and Services Implementation

We implemented a prototype of our infrastructure in the Linux kernel. The shared arena has been implemented by allocating a resident memory page for each application that uses our mechanisms and sharing it between the threads library and the kernel. This page is used by the threads library to inform the kernel on the number of processors each application can effectively use. In addition, the kernel informs the threads library on the number of processors it has granted to each application, the number of its preempted threads and the states of its threads.

A new scheduler has been implemented, in order to apply the desired scheduling policies to the applications that use our infrastructure. For the purposes of this work we use a policy that mixes time- and space-sharing to equalize the CPU time allocated to each program in the long-term, while letting parallel programs utilize multiple processors [12]. Our scheduler ensures that all applications make progress at the same time and no application starves for CPU time. The decisions of our scheduler are taken into account by the native scheduler only at specific points. More specifically, the native operating system scheduler selects the next thread to run on a processor. If the chosen thread belongs to an application which is controlled by our scheduler we assign the processor to a thread of the same application selected by our scheduler. In this way, the native scheduler continues to share fairly CPU time between all applications running on the system, while our scheduler applies the desired policy to adaptive applications.

Two more services have been implemented to assist applications in effectively utilizing idle processors. The first one comprises a mechanism which a thread can use to voluntarily handoff a processor to another thread. This mechanism can be used at thread joining. Each thread that completes its assigned computation checks the shared arena to see if there are preempted threads of the same application. If this is the case, it hands off the processor to one of them. The second mechanism handles yielding of processors. A thread can yield its processor to the operating system if it decides that it cannot use that processor effectively. If such a thread belongs to an application that uses our infrastructure, *local scheduling* is initiated, which means that a new scheduling decision affecting only that processor will be taken. The second-level scheduler tries to assign the processor to another thread of the same application that has useful work to execute. If such a thread does not exist, the processor is assigned to a thread of another application [8]. Local scheduling is also performed if a thread controlled by our scheduler is dequeued from the run-queue of the native scheduler. Such a thread cannot use a processor until it is re-inserted in the run-queue.

4 Threads Library Modifications

Only minor modifications are required in threads libraries, in order to exploit the functionality of the kernel-level infrastructure described earlier. The threads library has to invoke a *registration* system call before it starts creating threads.

In practically all implementations of dynamic shared libraries, there exists a function that initializes the library, which is called at the beginning of execution of each application and can be used for this purpose. The second addition has to be made in the function that implements joining of threads. A check is performed in the shared arena to examine if the application, to which the joining thread belongs, has preempted threads. If this is the case, a second system call is used to handoff the processor to another thread in the application.

No modifications are required in the threads library in order to exploit the functionality offered for threads that yield their processor. The required functionality is embedded in the native system call that implements processor yielding and is activated in the case of registered applications.

As an example, we describe the modifications applied to the LinuxThreads library, which is a POSIX-like threads library for Linux based systems. In Linux, the GNU compiler and linker provide the option of *constructor* functions. These are functions that are executed on behalf of the application before the execution of `main()`. The LinuxThreads library uses such a function to initialize its internal data for each application that starts executing on the system. The invocation of the registration system call is added to that function.

The function that implements joining of threads in a POSIX compliant library is called `pthread_join()`. The check in the shared arena and the invocation of the handoff system call are added, in our case, in this function. The overall changes required in any threads library are minimal and in the case of the LinuxThreads library the additional code is only about 30 lines long.

The yielding functionality of our infrastructure is indirectly exploited by the `pthread_mutex_lock()` and `pthread_cond_wait()` functions. The first implements locks and yields its processor when a thread fails several times to grant a lock, while the second yields its processor when a thread reaches a barrier.

5 Performance Evaluation

To evaluate the efficiency of our approach we used a set of applications written with two different programming models, namely OpenMP and Cilk. These programming models differ in all aspects, including scheduling, synchronization, and most notably target application domain. OpenMP is nowadays the *de facto* standard for portable shared-memory parallel programming in FORTRAN, C and C++. Cilk is an algorithmic multithreaded language that achieves optimal scheduling bounds for a certain class of multithreaded computations, i.e. multithreaded computations expressed with recursive fork/join sequences and having unit-distance data-dependencies between their threads.

The first application used in our experiments is Heat, which is distributed as an example program together with the Cilk programming language. Heat uses a Jacobi-type iterative algorithm to solve parabolic partial differential equations using a finite-difference approximation that models the heat diffusion problem. As representatives of the OpenMP programming model, we have chosen four applications from the NAS Benchmark Suite [5]. The Embarrassingly Parallel (EP)

benchmark generates pairs of Gaussian random deviates. The CG benchmark is an implementation of the Conjugate Gradient method, which performs unstructured grid computations and communications. MG uses a MultiGrid method to compute the solution of the 3-dimensional scalar Poisson equation. Finally, SP solves a 3-dimensional Scalar Pentadiagonal system. These applications were chosen because they mimic the data movement and computations of a wide range of real-world applications, with the exception of EP, which is used to determine the peak performance that a system can achieve.

We used version 5.3.1 of Cilk to compile Heat. For the applications selected from the NAS benchmarks suite we used the OmniMP [13] compiler, version 1.2s. Both environments convert programs written in the corresponding programming model into equivalent C programs that use POSIX threads. The executables are created from the intermediate code using the native C compiler of the system, which in our case is gcc 2.95.2.

The machine used for the evaluation is a Compaq Proliant 5500, equipped with 512 MBytes of physical memory and four Pentium-Pro processors, each one clocked at 200 MHz and incorporating 512 Kbytes of L2 cache. The operating system used is Linux 2.2.15 and the LinuxThreads library version is 2.1.3.

The first set of experiments consists of homogeneous workloads, i.e. workloads of concurrently running, identical copies of the same benchmark. Each benchmark creates four threads and the quantity of concurrently active benchmarks is equal to the degree of multiprogramming we want to achieve. We have experimented with multiprogramming degrees from 1 to 16 in powers of two. Each workload has been executed three times and the results reported are the averages of the three executions. We measure the average turnaround time of the benchmarks, which is a metric that characterizes the sustained throughput. The workloads used exceed the CPU capacity of the system by as much as a factor of four, which we consider as heavy load and representative for state-of-the-art SMP servers. We have used workloads with a total memory request less than the amount of memory available in the system, to isolate the impact of the schedulers on resident CPU-bound parallel jobs and factor out the adverse effects of paging.

The results are depicted in Figure 2. The absolute turnaround times of each benchmark have been normalized with respect to the absolute turnaround time of the same benchmark with 16-way multiprogramming and without using the kernel infrastructure. Table 1 lists the execution times used as normalization factors in each case. The maximum variation of run-times among the application instances within a workload is 2.96% for the modified and 2.4% for the unmodified kernel. The maximum variation of run-times among the repetitions of each experiment is 1.89% for the modified and 4.47% for the unmodified kernel. The differences are in both cases very small.

The execution times attained using the modified kernel and library are in most cases better and in the remaining few cases equal to the ones attained using the unmodified versions. A second observation is that embarrassingly parallel applications, like EP and SP, exhibit very good scalability in the presence

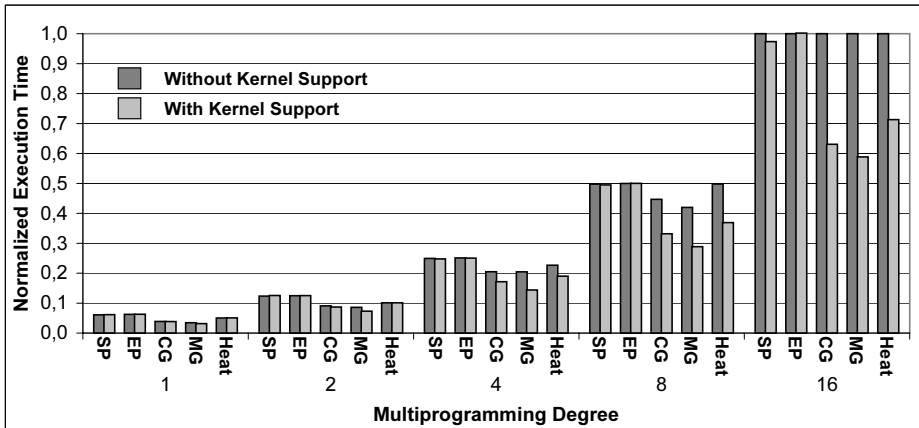


Fig. 2. Normalized turnaround times of the evaluation workloads

of multiprogramming, even without the modified versions of the kernel and the library. For such applications the execution times obtained by the modified versions are equal to the ones obtained by the standard versions. This indicates that the overhead of our infrastructure is negligible. The real value of our approach shows up in more complicated applications with irregular communication patterns and high synchronization requirements, like CG, MG and Heat. The system throughput improvement for these applications ranges from 4.6% to 41.2%. Moreover, using our services, the execution time increases linearly with respect to the increase of multiprogramming degree. This is not the case when the unmodified versions of the kernel and the library are used.

EP and SP have a coarse-grain barrier synchronization pattern, which facilitates fair allocation of CPU time among processes by the native scheduler. On the contrary, CG, MG and Heat synchronize more frequently. Under multiprogramming, the delay between the first and the last process arriving at a barrier is significantly increased. Our mechanisms reduce this delay by granting the processor to a process of the same application, whenever a yield occurs. The fairness of CPU time allocation among processes is preserved in the long term, via the native scheduler priority mechanism.

The second experiment consists of a mixed workload, which contains all five benchmarks. Each benchmark creates four threads and a copy of SP is the first

Table 1. Absolute turnaround times (in seconds) of all benchmarks with 16-way multiprogramming and without using the kernel infrastructure

SP	EP	CG	MG	HEAT
3392.61	570.64	351.90	274.61	306.87

that starts executing. After 15 seconds a copy of EP starts executing and finally, after another 15 seconds one copy of CG, MG and Heat start to execute simultaneously. The workload has been executed three times and the results, shown in Table 2, are the averages of the three executions. We measure the time that each application requires to complete its execution. The results show a significant improvement when the modified kernel is used, with the exception of Heat. This last result is unexpected, according to the execution time that each application requires individually. Further investigation showed that Heat never yields its processors, but its threads have the same probability with all other threads running in the system to take a processor that another application yields, when the unmodified kernel is used. This phenomenon raises an issue of unfairness for the native kernel. In the case of the modified kernel, the yielding system call tries first to hand off the processor to a thread of the same application. This reduces significantly the amount of CPU time allocated to the threads of Heat, thus improving the fairness of the system.

Table 2. Execution times (in seconds) of all benchmarks in the mixed workload

	CG	MG	HEAT	EP	SP
With kernel support	42.99	47.57	61.13	105.74	271.47
Without kernel support	95.42	77.93	54.12	121.39	289.51
Difference (%)	54.95	38.96	-12.96	12.89	6.23

6 Conclusions

We presented the design and implementation of a kernel-level infrastructure that provides a set of services to embed multiprogramming adaptability to thread-based systems in a totally transparent manner. The main advantages of this infrastructure are efficiency, transparency and non-intrusiveness. These services can be exploited by any threads library with limited modifications and applications benefit from our mechanisms and scheduling policies, without changing or recompiling their source code.

Our results have shown that applications conducting unstructured computations benefit greatly from the added functionality and that the performance gain increases when the multiprogramming degree is increased. Moreover, fully parallel applications that exhibit good scalability in the presence of multiprogramming without the modified versions of the kernel and the library, are not negatively affected by our mechanisms.

Currently, our implementation is focused on CPU bound applications. Our next step is to fine-tune the implemented services, in order to make them applicable to I/O bound applications. This will enable commercial applications with high I/O activity (e.g. databases, web servers etc.) to exploit our infrastructure.

Acknowledgments

This work has been supported by the Hellenic General Secretariat of Research and Technology (G.S.R.T.) research program 99ΕΔ-566.

References

1. N. Arora, R. Blumofe, and G. Plaxton. Thread scheduling for Multiprogrammed Multiprocessors. In *Proc. of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 1998. 515
2. M. Frigo, C. Leiserson, and K. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998. 514
3. A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proc. of the 1991 ACM Conference on Measurement and Modeling of Computer Systems*, pages 120–132, San Diego, USA, May 1991. 515
4. IEEE. *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996 edition, 1996. 514
5. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, 1999. 519
6. L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997. 515
7. X. Leroy. The LinuxThreads library home page. <http://pauillac.inria.fr/~xleroy/linuxthreads/index.html>. 516
8. D. Nikolopoulos, C. Antonopoulos, I. Venetis, P. Hadjidoukas, E. Polychronopoulos, and T. Papatheodorou. Achieving Multiprogramming Scalability on Intel SMP Platforms: Nanothreading in the Linux Kernel. In *Proc. of the 1999 Parallel Computing Conference*, pages 623–630, Delft, The Netherlands, August 1999. 516, 518
9. OpenMP A. R. B. *OpenMP Fortran Application Program Interface, Version 2.0*, 2000 edition, November 2000. 515
10. D. Papadopoulos. Hood: A User-Level Thread Library for Multiprogrammed Multiprocessors. Master's thesis, Department of Computer Sciences, University of Texas at Austin, August 1998. 515
11. C. Polychronopoulos, N. Bitar, and S. Kleiman. Nanothreads: A User-Level Threads Architecture. Technical report 1297, CSRD, University of Illinois at Urbana-Champaign, 1993. 514, 515, 517
12. E. Polychronopoulos, D. Nikolopoulos, T. Papatheodorou, N. Navarro, and X. Martorell. An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors. In *Proc. of the 12th Int. Conference on Parallel and Distributed Computing Systems*, USA, August 1999. 518
13. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP*, pages 32–39, Lund, September 1999. 520

14. A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *Proc. of the 12th ACM Symposium on Operating System Principles*, Litchfield Pk., USA, December 1989. 515