Achieving Multiprogramming Scalability of Parallel Programs on Intel SMP Platforms: Nanothreading in the Linux Kernel

Dimitrios S. Nikolopoulos, Christos D. Antonopoulos,
Ioannis E. Venetis, Panagiotis E. Hadjidoukas,
Eleftherios D. Polychronopoulos, 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

This paper presents the design and implementation of a *nanothreading* interface in the kernel of the Linux operating system for Intel Architecture-based symmetric multiprocessors. The objective of the nanothreading interface is to achieve robust performance of multithreaded programs and increased throughput in multiprogrammed shared memory multiprocessors, where multiple parallel and sequential programs with diverge characteristics and resource requirements execute simultaneously. The interface lets a multithreading runtime system and the kernel exchange critical scheduling information through loads and stores in shared memory, in order to enable parallel programs to adapt to dynamically changing resources and minimize their idle time. The same interface enhances the capability of the kernel scheduler to allocate resources evenly between competing programs. We discuss the main design and implementation issues concerning the nanothreading interface and provide experimental evidence which substantiates the efficiency of our implementation.

1 Introduction

Small-scale symmetric multiprocessors (SMPs) based on commodity Intel microprocessors are widely adopted as high performance and cost-effective compute servers. Proprietary and freeware operating systems for IA-based servers are SMP-compliant and provide operating system support for multithreading, typically implemented on top of kernel-level execution vehicles (EVs)[a] that share the same address space. At the same time, SMP-compliant operating systems support transparent multiprogramming, through time and space sharing of the system processors and memory.

The integration of multithreading with multiprogramming has been a hot spot in high performance computing research during the last decade [6]. Since modern SMPs are heavily multiprogrammed, the need for achieving scalability of parallel programs under multiprogramming is intensified[b]. The burden of

[a] We use the more general term *execution vehicle* instead of the term *kernel thread*, to cope with the inconsistencies in threads terminology between different operating systems.

[b] In this paper the term *scalability* is used in a broad sense, to characterize the ability of a

effectively integrating multithreaded programs in multiprogrammed environments is shared between the operating system and the runtime system layers. Unfortunately, most multithreading runtime systems are oblivious of multiprogramming and most SMP-compliant operating systems are oblivious of the fine-grain interactions between threads in multithreaded programs. This lack of coordination between runtime systems and operating systems has proven to be severely harmful for the performance of both individual applications and systems as a whole.

This work addresses the problem of integrating parallel programs in multiprogrammed environments, via the use of a lightweight shared memory interface, called the *nanothreading interface*[7], which lies between a multithreading runtime system and the operating system kernel. The nanothreading interface implements a communication infrastructure that enables a multithreaded program to automatically and transparently adapt to changes of the system resources allocated to it from the operating system at runtime. Adaptability is attained by matching the granularity of user-level threads to the number of processors available to the program at runtime and resuming maliciously preempted threads that execute on the critical path of the parallel computation. The same interface, used in the opposite direction, assists the kernel scheduler to apply sophisticated resource distribution strategies, by taking into account the actual resource requirements and the exploitable parallelism of each parallel program. The kernel gracefully grants parallel programs the authorization to use their EVs in the most effective way, while it maintains the responsibility of distributing system resources evenly. Put simply, the kernel acts like an advisory rather than as an explicit coordinator of the concurrency of parallel programs. The nanothreading interface architecture is adaptable and extensible, as it is not based on operating system or architecture-specific mechanisms. It thus promises to be a viable alternative for the effective integration of multiprocessing and multiprogramming in contemporary operating systems.

We present the main design and practical issues regarding the implementation of the nanothreading interface in the Linux operating system. These include the shared memory interface, mechanisms for processor allocation and affinity scheduling of nanothreaded jobs, mechanisms for fast resuming of maliciously preempted threads and the integration of the nanothreading jobs scheduler with the native time-sharing kernel scheduler. Our working prototype runs in Linux version 2.0.36 and is integrated with a user-level threads library that uses compiler knowledge to match the number of running user-level threads to the number of EVs granted to the program by the kernel [3]. We provide

parallel program to sustain robust and predictable performance when the load of the system due to multiprogramming is increased.

results from experiments with multiprogrammed workloads consisting of parallel jobs. The results show that the nanothreading Linux kernel achieves solid throughput improvements of up to 41% compared to the native Linux kernel.

The rest of this paper is organized as follows: Section 2 overviews the design and implementation of the nanothreading kernel interface. Section 3 provides experimental results and Section 4 discusses related and ongoing work.

2 Nanothreading Kernel Interface Design and Implementation

Three design principles guide the implementation of the nanothreading kernel interface. The first principle is that parallel programs should communicate with the kernel through loads and stores in shared memory in an asynchronous manner, since shared memory is the most efficient communication medium on a SMP. The second principle is that each parallel program should be armed with mechanisms that assist the program to effectively utilize the available processors and execute through the critical path, in the presence of preemptions of EVs from the operating system and blocking system calls. The third design principle is that the scheduler of the adaptive nanothreading programs should be seamlessly integrated with the native time-sharing scheduler of the operating system.

The nanothreading shared memory interface is organized around a *shared arena* [2,7]. Each multithreaded program reserves a portion of memory in its address space, which is shared between the program and the kernel. Care is taken so that the shared arena is situated in a single memory page. The pages containing shared arenas are pinned to physical memory in order to avoid paging out critical scheduling information at runtime. The shared arena is logically divided into a read-only and a read-write region. In the read-write region, each multithreaded program stores requests for processors, which are guided from the degree of parallelism that the program can effectively exploit and may vary during execution. In the same region, the program designates its EVs as *workers*, or *idlers*, depending on whether the EVs have user-level threads to execute in their run queues or not.

The processor requirements of a parallel program can be different from the number of processors allocated from the operating system to the program at runtime. The instantaneous number of physical processors allocated to each program is kept up-to-date from the kernel, which stores the associated information in the read-only region of the shared arena. A nanothreading program retrieves snapshots of this field by polling the shared arena whenever it initiates a parallel execution phase. In this way, the program is able to arrange its parallelism by creating as many user-level threads as the actual

instantaneous number of physical processors allocated to it from the operating system. The read-only region is also used by the kernel in order to communicate the number of undesirably preempted EVs to the program.

A virtual memory page in the Intel x86 architecture can be either read-only or read-write. In order to achieve both goals of using one memory page per application for the shared arena and protecting read-only information from accidental overwriting, the kernel keeps a private copy of the shared arena page and trusts only the data residing on it. The read-write region of the kernel copy is updated each time the application loses control of a processor. The read-only region of the application copy is updated when the operating system scheduler selects an EV belonging to that application to be given a processor for the next time quantum. The application copy is updated with a copy-on-write strategy to avoid some expensive TLB flushes.

The disposition of the shared arena page in the application's virtual address space is communicated to the kernel via a system call. This system call informs the kernel that the application uses the nanothreading interface and serves as a request to the kernel to create as many EVs as the application expects to use during its execution lifetime. All EVs needed are created at once, using a modified version of the native kernel code for cloning. This reduces the overhead of multiple system calls and results to significantly faster EV creation. The return address in the kernel stack of each created EV is set to point to the function the EV must upcall to when running for the first time i.e. the threads runtime library scheduler loop. The newly created EVs are not unblocked until the kernel decides to grant some processors to the application.

The assignment of runnable EVs to applications is a duty of the nanothreads kernel-level scheduler, which is invoked in four cases: when a nanothreading application leaves or enters the system, when an application changes its processor requirements and upon expiration of the nanothreads scheduler time quantum. The scheduling takes place in three phases. During the first phase, the scheduler decides how many runnable EVs will be assigned to each nanothreading application. A dynamic space sharing policy is used for this purpose [8]. Each program is granted processors according to the number of processors it requests and the overall system load. The second phase results to an indirect assignment of physical processors to the nanothreading applications selected in the first phase. More specifically, the nanothreading applications are granted the right to compete with other, non-nanothreading applications, for some specific processors. This strategy is used to integrate the nanothreading jobs scheduler with the native time-sharing scheduler of the operating system. In other words, the nanothreading jobs scheduler is solely responsible for deciding how many and which specific processors should be allocated

to each parallel program. Granting actual CPU time to the programs and managing priorities is a responsibility of the native time-sharing scheduler of the operating system. The major objective of the nanothreading scheduler in the second phase is to preserve the spatial and temporal locality of the processor sets in which each parallel program executes, via a strong affinity link maintained for each EV of a nanothreading program. In the final phase, a specific EV of an application chosen in the first phase is selected to serve as an EV during the next time quantum. A priority scheme is applied between the threads of each nanothreading application. Given a physical processor, if an EV was during the last time quantum executing on that physical processor, it is automatically selected. Otherwise the EVs that have previously been preempted while they were executing useful work, have the highest priority. The EVs that were previously preempted while idling constitute the next priority class. The two types of preempted EVs are distinguished with the worker/idler field in the shared arena. The EVs with the lowest priority are the ones that were previously voluntarily suspended at an idling point.

The nanothreading interface provides mechanisms to ensure that worker EVs which were undesirably preempted by the operating system scheduler are quickly resumed with application intervention. Each EV reaching an idling point of execution checks the shared arena for preempted worker EVs of the same application. If such EVs are found, the currently executing EV handoffs its processor in favor of a preempted worker, via a system call. EVs that find themselves idling for long also yield their processor in favor of an EV belonging to another nanothreading application, with the prospect that this EV will utilize the processor better. With this technique, nanothreading programs cooperate with the kernel to increase system throughput.

A common problem of user-level thread libraries is that when a user-level thread blocks in the kernel, e.g. while waiting for I/O, the corresponding EV also blocks. However, the user-level library has no means to be informed and activate another EV in order to keep using the processor efficiently. This can result to low processor utilization. In our implementation, when the kernel detects that an active EV of a nanothreading application is blocked, local scheduling, i.e. a scheduling having effect only on the processor assigned to the blocked EV, takes place. This leads to the selection and resumption of another EV. When the blocked EV is unblocked, it is either immediately resumed, or marked as a high priority preempted EV in the shared arena.

Several kernel services were added in the Linux kernel to support an efficient implementation of the nanothreading interface, including binding and unbinding of EVs to or from physical processors and an implementation of share groups for handling prematurely terminated EVs. The user-level side of

the nanothreading interface, was implemented in a research prototype of the Nanothreads runtime library [3], customized for Linux. Details on the implementation can be found in an extended version of this paper [5].

3 Performance Evaluation

We used a Quad Pentium Pro for the evaluation of the nanothreading Linux kernel. Each processor was clocked at 200 MHz and equipped with 512 Kbytes of L2 cache, while the total physical memory of the system was 512 Mbytes. We used the Pentium Pro hardware counter to measure time in our experiments.

The overhead for cloning EVs in the nanothreading kernel is 26 μs, which is about 4 times faster than the native Linux cloning overhead. Blocking and unblocking EVs from physical processors cost from 1.4 to 8 μs, depending on the previous state of the EVs. A full handoff from an idler EV to a worker EV costs 54 μs. In general, the nanothreading kernel services pose minimal system overhead, which is comparable to the overhead of well tuned user-level lightweight thread libraries.

We evaluated the overall performance of the nanothreading kernel in terms of the throughput achieved by the kernel scheduler for multiprogrammed workloads consisting of several multithreaded programs. We selected four applications from the SPLASH-2 suite [9], LU, FFT, Raytrace and Volrend as benchmarks. These applications either follow a task queue execution paradigm, or constitute of parallel regions separated from each other with global barriers. The changes needed in these applications in order to use the nanothreading interface were minor. The workloads used consisted of 1, 2, 4 and 8 identical copies of each application. All copies requested all processors of the system to execute, therefore the degree of multiprogramming was always equal to the number of copies of the program used in the workload. For each workload, we measured the throughput in terms of average turnaround time for our nanothreading kernel and the native Linux kernel (version 2.0.36) with the LinuxThreads POSIX 1003.1c-compliant threads package.

The results are depicted in Figure 1. Two observations are worth commenting in the charts. First, we witness that the nanothreading kernel demonstrates a solid improvement of throughput compared to the native Linux kernel. The increase of system throughput ranges from 9% to 41% with an average of 18% over all the experiments. Second, the throughput improvement achieved by the nanothreading kernel tends to magnify as the degree of multiprogramming and the load of the system increase. The improvements are mainly attributed to the effectiveness of the handoff strategy and the strong affinity scheduling mechanism which is employed in our nanothreading interface.

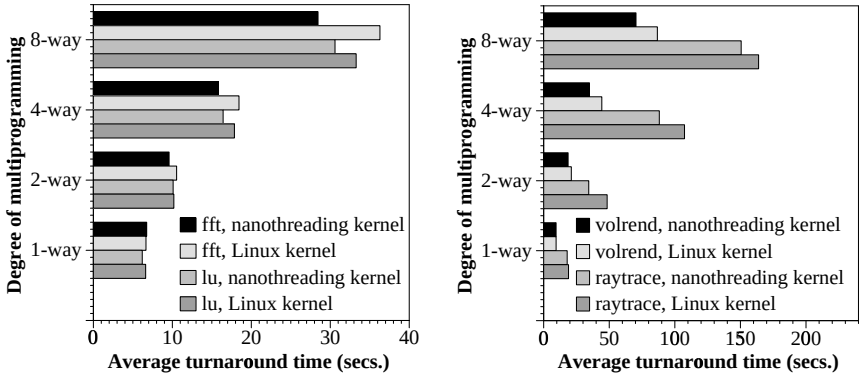

Figure 1: Results from executions of multiprogrammed workloads of SPLASH-2 benchmarks with the native and the nanothreading Linux SMP kernel.

4 Related and Ongoing Work

The idea of interfacing the user-level and kernel-level schedulers in order to provide adaptability of parallel programs in multiprogrammed environments is not new and has been used in the context of dynamic space-sharing processor scheduling strategies [1,4,10]. Our approach differentiates from these works in three critical aspects. Unlike previous implementations that relied on coarse-grain mechanisms like signals and upcalls, we implement the communication path between the runtime system and the kernel using just loads and stores in shared memory. We rely on a completely asynchronous mechanism based on polling to realize user-kernel communication. This mechanism minimizes the communication overhead and provides adequate means to parallel programs in order to readapt in the presence of frequent changes of system load and fine-grain interactions between threads. Second, the nanothreading interface is oriented towards providing efficient mechanisms to speedup the execution of parallel programs in the presence of undesirable preemptions from the operating system. The nanothreading interface as such is not coupled with a specific kernel scheduling policy. It rather enhances any scheduling policy with mechanisms for efficient multiprogrammed execution of parallel programs. Third, the nanothreading interface is seamlessly integrated with a UNIX time-sharing scheduler and enables efficient simultaneous execution of both nanothreading and non-nanothreading jobs. Integration of sophisticated parallel job scheduling strategies with time-sharing has not attracted considerable attention in previous research works on multiprocessor scheduling.

Our current efforts focus on integrating the nanothreading interface with the POSIX threads standard in order to extend its applicability to out-of-core multithreaded programs, including networking applications and Java. We also

investigate the integration of the nanothreading interface with the OpenMP standard for shared memory multiprocessing, as an infrastructure for implementing dynamic parallelism.

Acknowledgments

We are grateful to Constantine Polychronopoulos, David Craig and our partners in the NANOS project. This work is supported by the European Commission, through the ESPRIT IV Project No. 21907 (NANOS).

References

1. T. Anderson et.al. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Trans. on Computer Systems, 10(1), pp. 53–79, 1992.
2. D. Craig. *An Integrated Kernel-Level and User-Level Paradigm for Efficient Multiprogramming.* Master's Thesis, CSRD Technical Report 1533, University of Illinois at Urbana-Champaign, 1999.
3. X. Martorell et.al. *A Library Implementation of the Nanothreads Programming Model.* Proc. of Euro-Par'96, pp. 644–649, 1996.
4. C. McCann, R. Vaswani and J. Zahorjan. *A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors.* ACM Trans. on Computer Systems, 11(2), pp. 146–178, 1993.
5. D. Nikolopoulos et.al. *Achieving Multiprogramming Scalability of Parallel Programs on Intel SMP Platforms. Nanothreading in the Linux Kernel.* Technical Report HPCLAB-021298, Available at http://www.hpclab.ceid.upatras.gr, 1998.
6. C. Polychronopoulos. *Multiprocessing vs. Multiprogramming.* Proc. of the 1989 Int. Conf. on Parallel Processing, pp. II-223–II-230, 1989.
7. C. Polychronopoulos, N. Bitar and S. Kleiman. *Nanothreads: A User-Level Threads Architecture.* CSRD Technical Report 1297, University of Illinois at Urbana-Champaign, 1993.
8. E. Polychronopoulos et.al. *An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors.* Proc. of the 12th ISCA Int. Conf. on Parallel and Distr. Computing Systems, 1999.
9. S. Woo et.al. *The SPLASH-2 Programs: Characterization and Methodological Considerations.* Proc. of the 22nd Annual Int. Symposium on Computer Architecture, pp. 24–36, 1995.
10. K. Yue and D. Lilja. *An Effective Processor Allocation Strategy for Multiprogrammed Shared Memory Multiprocessors.* IEEE Trans. on Parallel and Distributed Systems, 8(12), pp. 1246–1258, 1997.