Thread Scheduling for Multi-Core Platforms

Mohan Rajagopalan Brian T. Lewis Todd A. Anderson
Programming Systems Lab, Intel Corporation
Santa Clara, CA 94054
{*mohan.rajagopalan, brian.t.lewis, todd.a.anderson*}*@intel.com*

Abstract

As multi-core processors with tens or hundreds of cores begin to proliferate, system optimization issues once faced only by the high-performance computing (HPC) community will become important to all programmers. However, unlike with HPC, the focus of the multi-core programmer will be on programming productivity and portability as much as performance. We introduce in this paper a novel scheduling framework for multi-core processors that strikes a balance between control over the system and the level of abstraction. Our framework uses high-level information supplied by the user to guide thread scheduling and also, where necessary, gives the programmer fine control over thread placement.

1 Introduction

Single-threaded processor performance is becoming power limited, so processor architects are increasingly turning to multi-core designs to improve processor performance. Such chips include Intel's experimental TeraFlops processor [9] and Sun's UltraSPARC T1 processor [10]. This design trend points to future systems having tens to hundreds of cores per processor, each of which is capable of running one or more software threads simultaneously. Experiments show that a wide variety of programs can benefit from the hardware parallelism [14] that such *many-core processors* (large-scale multithreaded chip multiprocessors) will provide.

Many-core processors will bring to desktop computing power formerly seen only in HPC systems. HPC programmers have traditionally used explicit thread and data placement to achieve the best performance for their parallel applications—an approach that not only requires a deep understanding of the hardware architecture of the HPC system, but also requires careful tailoring of the program to that specific hardware. We expect the number of people writing parallel programs to increase significantly. However, portability and ease of programming are equally, if not more, important to the general programmer than performance. Since the architecture of many-core systems is still evolving, portability is needed to allow the same program to run well on different kinds of many-core systems. As a result, one challenge for mainstream many-core programming is to develop mechanisms that provide the ability to do HPC-style customization for performance but do not compromise portability and programmability.

There are a number of challenges involved in designing a portable and easy-to-use, yet high performance scheduling interface for many-core platforms. For example, it will be essential to reduce shared cache misses since the on-die caches of many-core processors will be relatively small for at least the next several years, and memory latencies have grown to several hundred cycles [7]. Similarly, choosing which threads run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core since simultaneous-multithreaded (SMT) cores share low-level hardware resources such as TLBs among all threads.

To achieve the best performance for their parallel applications, programmers have traditionally used explicit thread and data placement. Most thread library implementations provide support for *pinning threads* to assign threads to specific CPUs (i.e., hardware threads) and to restrict their migration [11, 6]. But doing this requires the programmer to have a deep understanding of the system's architecture, which significantly hampers program portability and programmability. Similar problems exist for language-based HPC approaches [3, 1, 4] that are based on specifying *locales* or *regions* for computation.

Another challenge is that the scheduling primitives must support a wide variety of parallelization requirements. Moreover, some applications need different scheduling strategies for different program phases. The threads of data-parallel applications, for example, are typically independent: they share only a small amount of data except at certain communication points. For these applications, distributing threads widely among the different CPUs is beneficial. On the other hand, the threads of array-based programs typically share data heavily, so scheduling the threads on nearby CPUs to share data in common caches provides the best performance.

This paper introduces a scheduling framework for many-core systems that tries to strike a balance between the level of abstraction and the amount of control over the underlying system. This framework is based on the concept of the *Related Thread ID* (*RTID*), which is used to identify a collection of related software threads to the thread scheduler in order to improve their runtime performance. For example, a group of threads might be given a common RTID because they share some data or lock. RTIDs provide a higher level of abstraction than traditional fixed thread-to-processor mappings. In addition, RTIDs allow the programmer to express various scheduling constraints such as whether threads should be gang scheduled. We are currently finishing an initial implementation of our RTID-based scheduling framework as an extension of our existing McRT many-core runtime system [14]. This paper introduces RTIDs, presents the RTID interface our framework provides to user-level code, and describes our initial approach to the RTID-based scheduler design.

The remainder of the paper is organized as follows. The next section describes characteristics of the many-core architectures that we target. We present a detailed description of our framework and its interface API in Section 3, then Section 4 discusses how our scheduler framework addresses a variety of design issues. This is followed in Section 5 by an overview of our scheduler's implementation. Section 6 describes related work. The last section concludes and discusses future directions.

2 Target architecture

The underlying processor architecture plays a significant role in thread scheduling. While the exact architecture of large-scale many-core processors is still evolving, we can list some general characteristics.

Many-core processors will have tens to hundreds of cores, each of which run application threads on some number of hardware threads (HWTs). Initially, each HWT will have a private L1 cache and each core will have a shared L2 cache shared by all HWTs. Processors are likely to eventually have multiple levels of private cache per HWT, as well as a shared last-level cache. However, the many-core L2 caches will tend to be significantly smaller than those of traditional SMP systems. In addition, the cores will be interconnected by a high-bandwidth, low-latency interconnect.

The architectural differences of many-core processors have a number of consequences. First, on-die communication will be fast: the latency to access data from a different HWT will be about two orders of magnitude smaller than in today's SMP systems. Also, the cache size for each HWT and each core will be one or two orders of magnitude smaller than the per-processor caches of today's SMP systems. These differences mean that parallelism will be relatively cheap as long as the required data (and instructions) are on-die. However, the design of the interconnection fabric will also significantly affect latency, so thread placement decisions will have a big impact on thread performance.

In our current scheduling framework, we target single many-core processors. We expect to extend our framework to multiple many-core processors in the future. In addition, we also assume initially that the processor is homogeneous: that all cores are identical. However, future processors may be heterogeneous. For example, some cores may be faster, or may have special support for vector processing or other computation. We expect to later extend our framework to support these heterogeneous processors. Finally, we assume for now that the processor provides a coherent shared memory model with a single shared address space although this may have NUMA characteristics due to hierarchical caches. We do not specifically address distributed address spaces although we may do so in the future.

3 The scheduling framework

The primary goals of our scheduling framework are to improve application throughput and overall system utilization. A secondary goal of the framework is to improve fairness so that each thread continues to make good forward progress. These goals sometimes conflict. For example, to balance system load, the placement framework may schedule threads to run on HWTs distant from shared data. Also, scheduling threads to run too closely (for example, packed onto adjacent HWTs) can actually hurt performance if cache conflicts become too frequent. Our framework treats the question of how to balance these goals as a policy decision and lets the user specify which goal has the highest priority.

The framework provides support for both user-guided explicit placement as well as automatic best-effort placement. Expert programmers who want to control exactly where threads are scheduled can use explicit placement functions to control where to run threads and allocate data. However, we expect most users to use automatic placement since this simplifies application development and allows the same program to run well on a range of different many-core systems. In this scheme, the programmer identifies closely-related threads–threads that share data–and the runtime will do its best to schedule the threads close together so that they share data through nearby shared caches. In addition, the framework also provides a mechanism for *gang scheduling* threads—running them at the same time as well as close together. This feature is useful, for example, in programs where threads frequently acquire and release shared locks. In this case, latency will suffer if threads made newly-runnable must wait until they can be scheduled to run on a CPU.

At the heart of our framework is a configurable scheduler that decides when and the HWT on which each thread will be run. Our scheduling algorithm is tailored for efficiently running one or more applications using automatic placement. This approach enables both performance and portability since architecture-specific optimizations are implemented within the scheduler. If the underlying architecture changes, the existing scheduler can be replaced with one that is tailored for the new architecture.

3.1 RTIDs

RTIDs provide a mechanism for programmers to identify groups of threads that should run close together, for example, because they share common data. As an example, RTIDs can be used to schedule threads so that they can share data through local caches. The scheduler tries its best to run threads that share an RTID on nearby HWTs or cores. In our current design, each thread can have at most one RTID. If a thread has an RTID, it is specified when the thread is created. Threads without RTIDs can be scheduled to run anywhere.

Our framework also allows a programmer to directly associate an RTID with a data address. The scheduler will then try to run threads with that RTID close to each other and to the specified data: that is, it will try to run the threads on HWTs that share fast caches in order to minimize cache misses.

Finally, the framework supports attributes for RTIDs that supply additional information to help guide how threads sharing that RTID are scheduled. The first attribute is the expected number of simultaneous threads sharing the RTID. The scheduler will use this information to select (at least initially) which HWTs to use for the RTID's threads. Two other attributes specify whether to gang schedule the RTID's threads, and, if so, the minimum number of threads in the gang. These values allow the scheduler to decide when a quorum of the RTID's threads is ready to run. A fourth attribute allows the programmer to specify whether load balancing is more important than data proximity. This can be used, for example, to help optimize scheduling for data-parallel algorithms, where threads are mostly independent and communicate infrequently. This attribute can also be used to help spread out threads sharing an RTID so as to avoid cache conflicts.

Because a program's requirements can change over its lifetime, programs can also dynamically change an RTID's attributes. For example, if a single phase in a program needs gang scheduling, the program can specify it for just that one phase.

```
struct McrtRtidS;
typedef struct McrtRtidS * McrtRtid;

typedef struct {
    unsigned doLoadBalancing;
    unsigned width;
    unsigned minGangCount;
} McrtRtidAttrs;

McrtRtid mcrtCreateRtid(McrtRtidAttrs *attrs);
void mcrtFreeRtid(McrtRtid rtid);
McrtThread *mcrtThreadCreate(McrtThreadFunc f,
                             void *args,
                             McrtRtid rtid);
void *mcrtAssociateRtidWithHwt(McrtRtid rtid,
                               int hwt);
void mcrtAssociateRtidWithData(McrtRtid rtid,
                               void *addr);
void *mcrtMallocForRtid(size_t size,
                        McrtRtid rtid);
```

Figure 1: The RTID placement interface

3.2 The RTID placement interface

This section describes the interface that our placement framework presents to applications. Although we present this framework as an extension to the threading and scheduling support of the McRT many-core runtime system [14], nearly the same interface could be added to other operating systems. For example, supporting these extensions on Linux would require minor modifications to the scheduler and the Native POSIX Threads Library (NPTL) [6].

Figure 1 describes our interface for specifying placement. The interface is relatively short and includes some RTID-related types, two RTID management functions, a thread creation function, and a few explicit data and thread placement functions.

- `McrtRtid` values, which are opaque, represent RTIDs.
- `McrtRtidAttrs` structures specify attributes for RTIDs. With a `McrtRtidAttrs` structure, a program can specify a number of items to either control or guide the scheduling of the RTID's threads. Most programs set the `doLoadBalancing` field to 0 to indicate that data proximity is more important than load balancing. However, this field can also be set 1 to have the scheduler emphasize load balancing. The `width` field gives the expected number of simultaneous threads sharing the RTID, or 0 if unknown. Finally, if `minGangCount` is greater than 1, the RTID's threads will be gang scheduled; otherwise its threads will be run near each other, but not necessarily at the same time.
- RTIDs are created using the `mcrtCreateRtid`

function. Note that its `attrs` argument is a pointer to an `McrtRtidAttrs` structure instead of the structure itself; this allows the program to update the RTID's attributes by simply modifying the structure in memory; the changes will take effect when the scheduler next runs.

- The `mcrtFreeRtid` function frees any resources associated with the RTID.
- The `mcrtThreadCreate` function creates a new thread. If the new thread should be associated with an RTID, that RTID is given by the `rtid` parameter; otherwise, 0 (i.e., `NULL`) should be given.
- Explicit placement support is provided by the `mcrtAssociateRtidWithHwt` function. This pins an RTID to a particular HWT so that its threads will always run on that hardware thread.
- Two functions control data placement. The first function, `mcrtAssociateRtidWithData`, indicates that the RTID's threads should be run on HWTs sharing fast caches that can hold data at or close to the given address. The second, `mcrtMallocForRtid`, is a variant of `malloc` that allocates the data "close" to the given RTID. It has one of the HWTs executing the RTID's threads allocate the data.

4 Discussion

This section summarizes how our framework addresses a number of issues that would be faced by any scheduler design. First, there is the question of what information the application programmer should provide to guide thread scheduling. Our approach is to require only high-level information unless explicit thread placement is being used. We expect the developer to identify the threads that share common data or locks and to assign those threads the same RTID. The scheduler will then assign HWTs to these threads in such a way as to reduce cache misses. To help guide the scheduler, we also allow the user to provide a small amount of additional information such as whether to emphasize load balancing or proximity to data, and whether gang scheduling is required. Because our scheduling framework does not require the developer to supply architecture-dependent information, such as the specific cache level threads should share, it helps to preserve the application portability that general-purpose developers need.

Another issue is whether to support explicit placement of threads and data. Although our scheduling framework is oriented towards automatic thread scheduling, we also support explicit placement by allowing the programmer to bind an RTID to a particular HWT. In this way, our API for explicit placement is consistent with that for automatic placement. However, the framework does not make any performance guarantees if programs mix explicit and automatic placement for their threads. In addition our framework also supports explicit data allocation by allowing the client to specify that data be placed close to an RTID. In this case the allocation is performed on an HWT that is executing threads associated with the RTID.

There is also the question of whether the framework can optimize thread scheduling on a range of different systems. In the past, parallel programs typically had to be customized for a particular system to get good scalability and performance on that system. This has been true, for example, for HPC programs. However, a program tuned to run well by itself on a particular system may not perform well if it is only one of several programs running simultaneously. Since general-purpose systems rarely run a single application at a time, we decided in favor of portability and ease of programming. Our approach is to have the developer guide placement using RTIDs and then have the scheduler use this information to place threads to achieve the best possible performance. While this approach may not be able to achieve the high performance that programs hand-tuned for a particular system might achieve, it should provide good performance across a range of programs on different architectures.

5 Scheduler design

Our framework uses an extension of the scheduler in McRT [14]. This is a highly configurable task-queue-based scheduler, and clients may specify the number of task-queues, the task-queue to HWT mapping, as well as the scheduling policies. Our initial implementation uses one task-queue per processor core and a combination of the scheduler's existing work-stealing and work-distribution scheduling policies. This configuration allows us to maximize sharing through the L2 (last-level) caches, while making use of the existing scheduling policies to achieve good load balancing. In general, our goal is to improve performance by minimizing contention for different resources and by maximizing HWT usage.

When new software threads are created, the scheduler tries to distribute them to task queues based on the load in the system. If the number of threads is less than the number of HWTs, the scheduler tries to improve throughput by channeling work to unused HWTs. Once every HWT is in use, the scheduler switches to *saturation mode* where scheduling is increasingly guided by RTIDs. When a thread is created in saturation mode, its RTID is compared to those of threads already in the system. If that RTID already exists, the scheduler will place this thread on a task queue for a core that shares as many levels of cache as possible with other threads sharing the

RTID. If that RTID is new, the thread is placed in the least loaded task-queue. The scheduler also uses work-stealing to try and keep the system's load balanced. Idle HWTs steal work from other tasks queues in the system. As a result, threads may migrate across HWTs. However, we expect thread migration to be relatively inexpensive on many-core processors.

As threads continue to be added to the system, eventually threads of multiple RTIDs will be scheduled on the same core. Our scheduler optimizes the order in which threads are run by grouping them according to RTID. When an RTID is scheduled, the threads within that group are run round-robin for a period of time. To ensure fairness, RTIDs are occasionally preempted and replaced at the end of the task queue. This approach is an extension to CPU affinity in which schedulers place threads on the same HWT they last ran on in order to reuse data in local caches. Although CPU affinity is likely to be less important on many-core systems, where the fast interconnect reduces the cost of an on-die memory miss, it is still likely to be important for many applications.

6 Related work

Many thread libraries on both Windows and POSIX operating systems give clients some control over thread scheduling. For example, POSIX threads allow clients to specify a scheduling policy and priority for their threads; standard scheduling policies include whether to use a first in-first out or round-robin scheduling policy. Furthermore, thread libraries typically provide some support for binding a thread to one or more processors.

Solaris provides a locality-oriented mechanism to optimize performance in NUMA systems. *lgroups* (locality groups) [5] represent a collection of CPUs and memory resources that are within some latency of each other. Lgroups are hierarchical and are created automatically by Solaris based on the system's configuration and different levels of locality. The system assigns each newly-created thread a home lgroup that is based on load averages, although applications can give a thread a different home lgroup. Lgroups help to control where threads and memory are allocated: when a thread is scheduled to run, it is assigned the available CPU nearest to the home lgroup, and memory is gotten either from the home lgroup or some parent lgroup. As a result, lgroups can be used to improve the locality of an application's threads, but are more restricted than RTIDs, which can also be used to specify gang scheduling and other scheduling-related information.

Since RTIDs are used to group related threads they may be compared with the *process groups* supported by both Windows and POSIX operating systems. Process groups allow a group of processes to be treated as a unit that can be identified by a *process group ID*. While process group IDs have some similarities to RTIDs, they are primarily used to control the distribution of signals and not specifically intended to improve performance.

Recent work on scheduling algorithms for many-core processors includes that of Fedorova [7]. Her cache-fair thread scheduling algorithm provides fairer thread schedules and greater performance stability on shared-cache multi-core processors by continually adjusting, for each thread, its time quantum to ensure that that thread has a fair miss rate in the L2 (i.e., last-level on-die) cache. The target-miss-rate algorithm helps to keep processor utilization high by achieving a target L2 cache miss rate. It does this by dynamically identifying threads that provoke the highest miss rates and lowering their priorities. Anderson, Calandrino, and Devi [2] apply a cache-aware thread co-scheduling algorithm to real-time systems. This algorithm reduces L2 contention by avoiding the simultaneous scheduling of problematic threads while still ensuring real-time constraints.

There is a large body of related work on thread scheduling to improve application performance on SMT processors. For example, Parekh, Eggers, Levy, and Lo [13] demonstrate significant performance speedups from scheduling algorithms that uses PMU feedback to select the best threads to schedule together. They found the best performance from an algorithm based on IPC feedback. Fedorova's non-work-conserving scheduler [8], which improves application performance by reducing contention for shared processor resources like the L2 (last-level on-die) cache. The algorithm uses an analytical model to dynamically determine when it makes sense to run fewer threads than the number of HWTs.

An especially relevant paper on SMT scheduling is that by Thekkath and Eggers [15], who found that placing threads sharing data on the same processor did not improve cache performance or execution time. However, their study did not consider more threads than HWTs and their threads were fairly coarse-grained. We would like to investigate whether their results still hold today for both many-core processors and emerging parallel applications in domains such as streaming, media processing and *RMS* (recognition, mining, and synthesis) applications [12].

7 Conclusion

Many-core systems will encourage the development of parallel, general-purpose applications. As the number of programmers for these applications increases, the need will grow for simpler but still effective ways of using large-scale parallelism. This paper describes a scheduling framework that tries to achieve good performance for the majority of applications on different many-core platforms without compromising ease of programming. We believe having applications provide high-level guidance to a scheduler through RTIDs will let the scheduler do a better job for most programs than if those programs did explicit thread placement.

We are in the process of completing the initial implementation of this framework. Several mechanisms such as thread stealing and support for multiple scheduling policies are already a part of the McRT runtime, and our framework implementation is an extension of the current McRT scheduler. Our near-term plans are to evaluate the framework when used to run a number of our benchmarks, and then to experiment with different scheduling policies.

Future work includes the addition of several features our scheduling framework does not currently support. This includes support for multiple many-core processors, non-homogeneous processors, thread priorities, hierarchical RTIDs, multiple RTIDs for each thread, and NUMA distributed address spaces. In addition, we also plan to investigate integrating different feedback-based mechanisms into our scheduler to try to improve performance, system utilization, and fairness. In particular, we want to study how we could use PMU-based information about thread and system behavior to augment the RTID information from users.

8 Acknowledgments

We want to thank Jim Stichnoth, Neal Glew, and Youfeng Wu for their help in preparing this paper. We also thank the anonymous referees for their helpful feedback and suggestions.

References

[1] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT, S. The Fortress language specification, version 1.0beta.

[2] ANDERSON, J. H., CALANDRINO, J. M., AND DEVI, U. C. Real-time scheduling on multicore platforms. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)* (Washington, DC, USA, 2006), pp. 179–190.

[3] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the Chapel language. *Int. Journal of High Performance Computing Applications* (Exp 2007).

[4] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05* (2005).

[5] CHEW, J. Memory placement optimization (MPO). http://opensolaris.org/os/community/% linebreak[0]performance/numa/mpo_update.pdf, Jul 2006.

[6] DREPPER, U., AND MOLNAR, I. The native POSIX thread library for Linux. http://people.redhat.com/drepper/nptl-design.pdf.

[7] FEDOROVA, A. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, Nov 2006.

[8] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proc. USENIX 2005 Annual Technical Conference* (Anaheim, California, April 2005).

[9] INTEL. Teraflops research chip. http://www.intel.com/research/platform/terascale/teraflops.htm, Feb 2007.

[10] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (Mar 2005), 21–29.

[11] LEWIS, B., AND BERG, D. J. Multithreaded programming with Pthreads. *Prentice Hall* (1998).

[12] LIANG, B., AND DUBEY, P. Recognition, mining and synthesis. *Intel Technology Journal 9*, 2 (May 2005).

[13] PAREKH, S., EGGERS, S., AND LEVY, H. Thread-sensitive scheduling for SMT processors. *University of Washington Technical Report 2000-04-02* (2000).

[14] SAHA, B., ADL-TABATABAI, A., GHULOUM, A., RAJAGOPALAN, M., HUDSON, R., PETERSEN, L., MENON, V., MURPHY, B., SHPEISMAN, T., SPRANGLE, E., ROHILLAH, A., CARMEAN, D., AND FANG, J. Enabling scalability and performance in a large scale CMP environment. *EuroSys* (March 2007).

[15] THEKKATH, R., AND EGGERS, S. J. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1994).