Volume 11 Issue 03 Published, August 22, 2007 ISSN 1535-864X DOI: 10.1535/itj.1103.05

Tera-scale Computing

Architectural Support for Fine-Grained Parallelism on Multi-core Architectures

Architectural Support for Fine-Grained Parallelism on Multi-core Architectures

Sanjeev Kumar, Corporate Technology Group, Intel Corporation
Christopher J. Hughes, Corporate Technology Group, Intel Corporation
Anthony Nguyen, Corporate Technology Group, Intel Corporation

Index words: multi-core, loop and task parallelism, architectural support

ABSTRACT

In order to harness the additional compute resources of future Multi-core Architectures (MCAs) with many cores, applications must expose their thread-level parallelism to the hardware. One common approach to doing this is to decompose a program into parallel "tasks" and allow an underlying software layer to schedule these tasks on different threads. Software task scheduling can provide good parallel performance as long as tasks are large compared to the software overhead. We examine a set of Recognition, Mining, and Synthesis (RMS) applications and find that a significant number have small tasks for which software task schedulers achieve only limited parallel speedups.

We propose a hardware technique to accelerate dynamic task scheduling on MCAs with many cores. We compare this hardware to highly tuned software task schedulers for a set of RMS benchmarks with small tasks. The proposed hardware delivers significant performance improvements over the best software scheduler: for 64 cores, it is 88% faster on a set of loop-parallel benchmarks and 98% faster on a set of task-parallel benchmarks.

INTRODUCTION

Multi-core Architectures (MCAs) provide applications with an opportunity to achieve much higher performance than uniprocessor systems. Furthermore, the number of cores on MCAs is likely to continue growing, increasing the performance potential of MCAs. However, realizing this performance potential in an application requires the application to expose a significant amount of thread-level parallelism.

A common approach to exploiting thread-level parallelism is to decompose each parallel section into a set of tasks. At runtime, an underlying library or run-time environment distributes (schedules) these tasks to the software threads [2, 3, 4]. To achieve maximum performance, especially in systems with many cores, it is desirable to create many more tasks than cores and to dynamically schedule the tasks. This allows for much better load balancing across the cores.

We examine a set of benchmarks from an important emerging application domain: Recognition, Mining, and Synthesis (RMS) [1]. Many RMS applications have very high compute demands and can therefore benefit from a large amount of acceleration. Further, they often have abundant thread-level parallelism. Thus, they are excellent targets for running on MCAs with many cores.

For previously studied applications and architectures, the overhead of software dynamic task schedulers is small compared to the size of the tasks, and therefore, enables sufficient scalability. However, we find that a significant number of RMS applications are dominated by parallel sections with small tasks. These tasks can complete execution in as few as 50 processor clock cycles. For these, the overhead of software dynamic task scheduling is large enough to limit parallel speedups.

We therefore propose a hardware technique to accelerate dynamic task scheduling on scalable MCAs. It consists of two components: (1) a set of hardware queues that cache tasks and implement task scheduling policies, and (2) per-core *task prefetchers* that hide the latency of accessing these hardware queues. This hardware is relatively simple, scalable, and delivers performance close to optimal.

We compare our hardware proposal to highly tuned software task schedulers, and also to an idealized hardware implementation of a dynamic task scheduler (i.e., operations are instantaneous). On a set of RMS benchmarks with small tasks, it provides large performance benefits over the software schedulers and gives performance very similar to the idealized implementation.

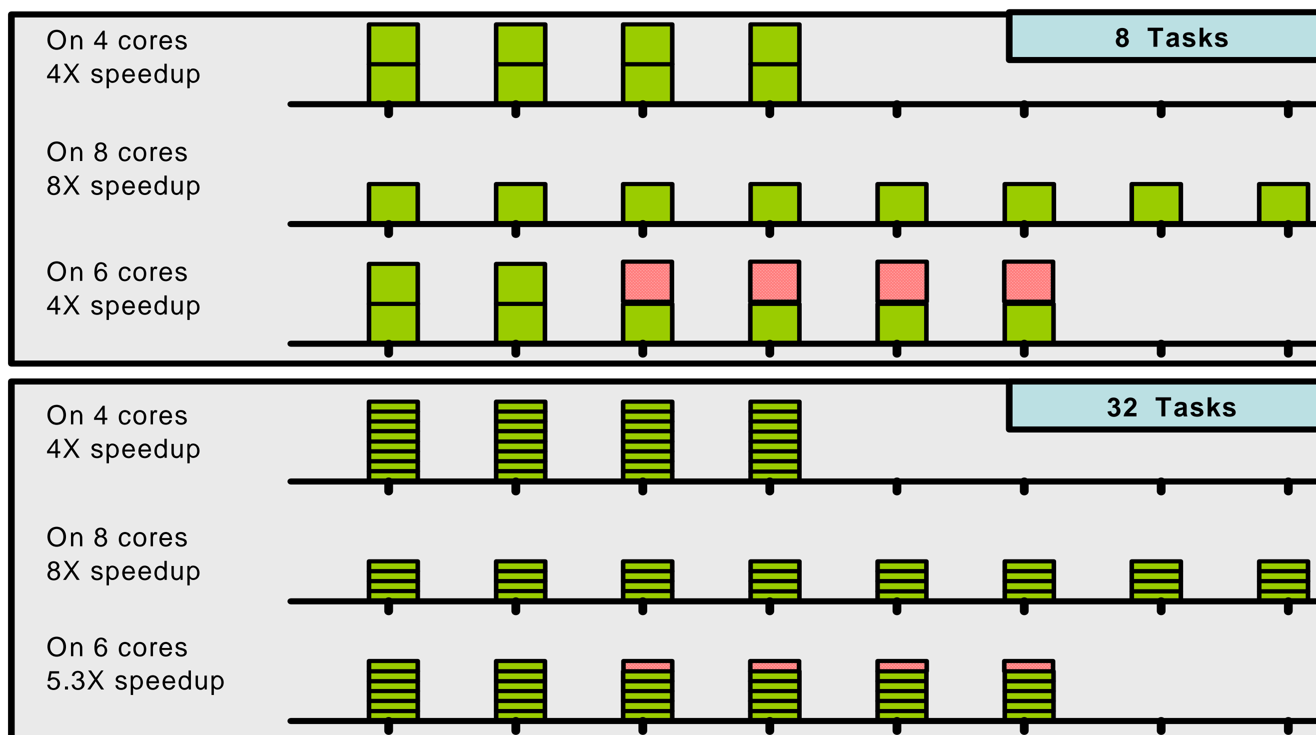

Figure 1: Impact of multiprogramming

Our contributions are as follows:

1. We make the case for efficient support for fine-grained parallelism on MCAs. Parallel tasks can be as fine as 50 processor clock cycles.
2. We propose a hardware scheme that provides architectural support for fine-grained parallelism. Our proposed solution has low hardware complexity and is fairly insensitive to access latency to the hardware queues.
3. We demonstrate that the proposed architectural support has significant performance benefits. First, it delivers much better performance than optimized software implementations: 88% and 98% faster on average for 64 cores on a set of loop-parallel and task-parallel RMS benchmarks, respectively. In addition, it delivers performance close to (about 3% on average) an idealized hardware implementation of a dynamic task scheduler (i.e., operations are instantaneous).

A CASE FOR FINE-GRAINED PARALLELISM

Previous work on dynamic load balancing targeted coarse-grained parallelism, i.e., parallel sections with either large tasks, a large number of tasks, or both. The target was primarily scientific applications for which this assumption is valid. For these applications, an optimized software implementation delivers good load balancing with an acceptable performance overhead.

The widespread trend towards an increasing number of cores becoming available on mainstream computers—both at homes and at server farms—motivates efficient support for fine-grained parallelism. Parallel applications for the mainstream are fundamentally different from parallel scientific applications that run on supercomputers and clusters in a number of aspects. We discuss these differences in detail in this section.

Architecture

Reduced communication overhead: MCAs dramatically reduce communication latency and increase bandwidth between cores. This allows parallelization of modules that could not previously be profitably parallelized.

Usage scenarios: These architectures are designed to be used with virtualization technologies as well as multiprogramming. In both these instances, the number of cores assigned to an application can change during the course of its execution. Maximizing the available parallelism under these conditions requires exploiting fine-grained parallelism.

Consider the example shown in Figure 1 that illustrates this using an 8-core MCA. It presents two scenarios where the parallel section is broken down into 8 and 32 equal-sized tasks (represented by green boxes). In a parallel section, if a core finishes its tasks before all other cores have finished their tasks, it has to wait. This results in wasted compute resources (shown in red). In each of the two scenarios, it shows the performance when varying number of cores are assigned to this parallel section. In both scenarios, with 4 and 8 cores, all the assigned cores are fully utilized. However, when 6 cores are assigned to the application, the first scenario wastes significant compute resources. In fact, it achieves the same speedup as when it was assigned 4 cores. In the second scenario, there are many fewer wasted compute resources because the parallel section was broken into finer-grained tasks.

This problem worsens when the number of cores increases. Figure 2 shows the maximum potential speedup on a 64-core MCA for a varying number of tasks. The ideal situation would be if the graph was linear, implying that each additional core would deliver additional performance. When only 64 tasks are used, the application would see no performance improvement even when the number of cores assigned to an application was increased from 32 to 63. To approach the ideal situation, one needs a much larger number of tasks (say 1024).

Performance portability across platforms: Parallel scientific computing applications are often optimized for a specific supercomputer to achieve the best possible performance. However, for mainstream parallel programs, it is much more important for the application to get good performance on a variety of platforms and configurations. This has a number of implications that require exposing parallelism at a finer granularity.

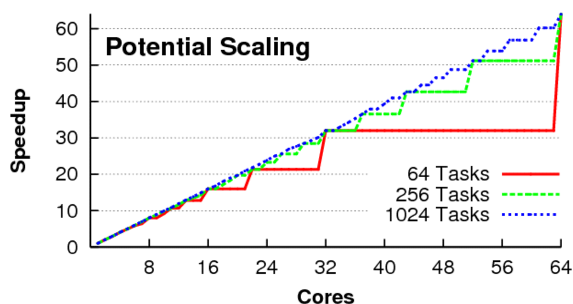

Figure 2: Theoretical scalability

First, the number of cores varies from platform to platform. For reasons similar to that for virtualization/multiprogramming, finer-granularity tasks are necessary.

Second, MCAs are likely to be asymmetric for a number of reasons including heterogeneous cores, Hyper-Threaded (HT) cores, Non-Uniform Cache Architecture (NUCA), and Non-Uniform Memory Architecture (NUMA). This means that the different threads on the core might progress at different rates. For instance, two threads sharing a core run at a different rate than two threads running on two different cores.

Figure 3 illustrates the impact of asymmetry with an example. Consider an application that breaks its parallel section into tasks that represent equal amounts of work (shown in green). However, asymmetry in architecture results in each task taking a different amount of time to complete. The result is wasted compute cycles (shown in red). This example shows that to ensure good performance in the presence of hardware asymmetry, it is best to expose parallelism at a fine grain.

Workloads

To understand emerging applications for multi-core architectures, we have parallelized and analyzed emerging applications (referred to as RMS [1]) from a wide range of areas including physical simulation for computer games as well as for movies, raytracing, computer vision, financial analytics, and image processing. These applications exhibit diverse characteristics. On the one hand, a number of modules in these applications have coarse-grained parallelism and are insensitive to a task queuing overhead. On the other hand, a significant number of modules have to be parallelized at a fine granularity to achieve reasonable performance scaling.

Recall that Amdahl's law dictates that the parallel scaling of an application is bounded by the serial portion. For instance, if 99% of an application is parallelized, the remaining 1% that is executed serially will limit the maximum scaling to around 39X on 64 threads.

This means that even small modules need to be parallelized to ensure good overall application scaling.

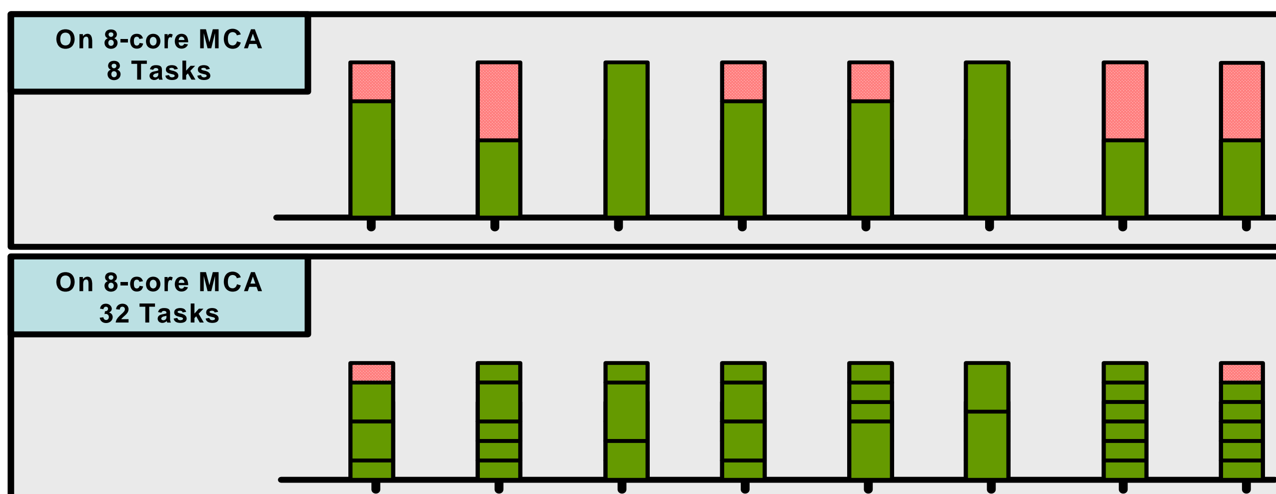

Figure 3: Impact of asymmetry in architecture

Ease of Programming

The use of modularity will continue to be very important for mainstream applications for several reasons. Modularity is essential to developing and maintaining complex software. In addition, applications are increasingly composed of software components from multiple vendors. These include middleware as well as libraries optimized for specific platforms.

Modular programs require writers of individual modules to make decisions about how best to parallelize that module. Consider a simple example where an application is composed of two modules: the main program and an optimized math library. Suppose that parallelizing either the library or the main program is sufficient to exploit all the parallel computing resources on the machine. However, modularity dictates that one module does not make assumptions about another module. This requires that for the best performance on a variety of platforms, both modules be parallelized in cases where the other module is not parallelized. The net result will be a finer granularity of parallelism in the application.

ARCHITECTURAL SUPPORT FOR FINE-GRAINED PARALLELISM

Software implementations of task queues incur an overhead (e.g., for enqueues and dequeues). This overhead grows with an increasing number of threads due to increased contention on shared data structures in the software implementation. Thus, if the tasks are small, the overhead can be a significant fraction of application execution time. This limits how fine-grained the tasks can be made and still achieve performance benefits with a large number of cores.

Therefore, we investigate adding hardware for MCAs that accelerates task queues. This hardware operates under the covers (i.e., is not visible to application writers) to accelerate the task queue operations that are key to high performance on many-core architectures. In particular, it provides very fast access to the storage for tasks. This includes performing fast task scheduling (i.e., determining which task a core should execute next). Its task scheduling is based on work stealing—a well-known scheduling algorithm.

Using the Proposed Hardware

Applications interface to the proposed hardware via a software library. This allows programmers to use the same intuitive API that they use for software implementations of task queues. Since the software library hides the proposed hardware from applications, only the library needs to directly interact with this hardware. Besides initialization and termination, the only operations that the library needs to perform are task enqueues and dequeues.

In current software task queue implementations, each task is represented as a tuple, a set of associated items, as shown in Figure 5. Typically, the tuple entries will be function pointers, jump labels, pointers to shared data, pointers to task-specific data, and iteration bounds, but they could be anything. An enqueue places a tuple into a software data structure for storage, and a dequeue retrieves a tuple from the data structure.

Our proposed hardware is primarily intended to accelerate enqueue and dequeue operations. Thus, the hardware stores tuples on enqueue operations and delivers tuples on dequeue operations. It does not interpret the contents of the tuples. This provides flexibility to the software library in that the library determines the meaning of each entry in a tuple. This flexibility allows the library writer to optimize for applications with different needs.

Our Proposed Hardware

We consider an MCA chip where the cores are connected to a cache hierarchy by an on-die network. The proposed hardware consists of two separate hardware components: a Local Task Unit (LTU) per core, and a single Global Task Unit (GTU). This is illustrated in Figure 4.

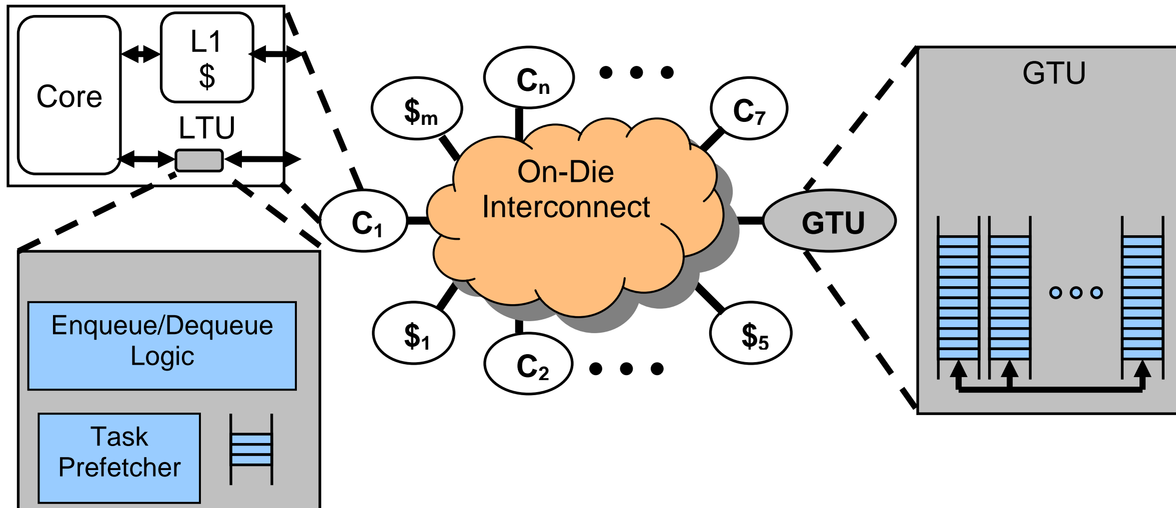

Figure 4: Our proposed hardware in a MCA chip with cores (C_i) and parts of the last-level shared cache ($\$_i$)

Global Task Unit (GTU)

The GTU holds enqueued tasks in a set of hardware queues. There is one hardware queue per logical core in the chip. This allows the use of the distributed task scheduling algorithm. The GTU also includes logic for implementing this algorithm. Since the hardware queues are physically close to each other, the proposed scheme can quickly determine which queues are empty and which are not. This makes stealing tasks much faster than for software implementations of distributed task scheduling. It also allows the hardware to quickly detect when all tasks are complete. This is important so that the main thread can start executing the serial code following the parallel section as quickly as possible.

Function pointer	Parameter 1	Parameter 2	Parameter 3

Figure 5: An example task tuple format

The GTU is physically centralized on the chip. Communication between the GTU and the cores is via the same on-die interconnect as the cache subsystem. The downside of a physically centralized GTU is that as the number of cores on a chip increases, the average communication latency between a core and the GTU also increases. This latency, if not hidden, could impact performance. Therefore, we address this with task prefetchers at each core, as described below.

The size of the queues in the GTU is bounded. When the queues are full, the hardware generates an exception. The exception handler can move some of the tasks from the hardware queues into memory creating room for future task enqueues. An underflow mechanism is used to move the overflown tasks back into hardware queues at a later point [2].

Multiprogramming is also supported by the hardware by using the same overflow and underflow mechanism to move tasks from hardware into memory, and vice versa, on context switches [2].

Local Task Unit (LTU)

Each core has a small piece of hardware to interface with the GTU, called the LTU. In addition to hardware for interfacing with the GTU, the LTU also contains a task prefetcher and small buffer to hide the latency of accessing the GTU. Hiding this latency can significantly improve performance. While a typical enqueue operation from a thread can be almost entirely overlapped with useful work, a dequeue operation is on a thread's critical path. If a logical core were to wait to contact the GTU

until the thread running on it finished its current task, the thread would have to stall for the entire GTU access latency. If the latency is significant compared to the size of a task, this can take up a significant fraction of an application's execution time. Therefore, the LTU tries to hide the dequeue latency. It does this by trying to keep at least one task in the LTU's buffer at all times. A dequeue is able to grab such a task very quickly. The LTU operates as follows.

- On a dequeue, if there is a task in the LTU's buffer, that task is returned to the thread and a prefetch for the next available task is sent to the GTU. When the GTU receives a prefetch request, it treats it as a regular dequeue, and will steal a task from another logical core's queue if necessary.
- On an enqueue, the task is placed in the LTU's buffer. Since the proposed hardware uses a LIFO ordering of tasks for a given thread, if the buffer is already full, the oldest task in the buffer is sent to the GTU.

In our experience, the LTU's buffer only needs to hold a single task to hide the GTU access latency. If this latency grows in the future, the buffer could be made larger. However, there is a cost: tasks in an LTU's buffer cannot be stolen since they are not visible to the GTU. This could hurt performance if there are only a few tasks available at a time.

Table 1: Loop-level benchmarks and their inputs

Benchmark	**Data set**
Gauss-Seidel	128x128, 256x256, 512x512
Dense Matrix-Matrix Multiply (MMM)	64x64, 128x128, 256x256
Dense Matrix-Vector Multiply (MVM)	64x64, 128x128, 256x256, 512x512
Sparse Matrix-Vector Multiply (MVM)	4 data sets
Scaled Vector Add	512, 1024, 4096, 16384 elements

Table 2: Task-level benchmarks and their inputs

Benchmark	**Data set**
Game Physics Constraint Solver	4 Models
Binomial Tree	512, 1024, 2048, 4096
Canny Edge Detection	cars, costumes, camera2, camera4
Cholesky Factorization	4 data sets
Forward Solve	4 data sets
Backward Solve	4 data sets

EXPERIMENTAL EVALUATION

Benchmarks

We evaluate our proposed hardware on benchmarks from a key emerging application domain: RMS. All benchmarks were parallelized within our lab. Table 1 and Table 2 give the benchmarks and their data sets.

Loop-level parallelism: We use primitive matrix operations and Gauss-Seidel as a set of benchmarks with loop-level parallelism since these are both very common in RMS applications and very useful for a wide range of problem sizes. Most of these benchmarks are standard operations and require little explanation. The sparse matrices are encoded in compressed row format. Gauss-Seidel iteratively solves a boundary value problem

with finite differencing using a red-black Gauss-Seidel algorithm. These benchmarks are straightforward to parallelize; each parallel loop simply specifies a range of indices and the granularity of tasks. We evaluate each benchmark with several problem sizes to show the sensitivity of performance to problem sizes.

Task-level parallelism: We use modules from full RMS applications as a set of benchmarks with task-level parallelism. Task-level parallelism is more general than loop-level parallelism where each parallel section starts with a set of initial tasks and any task may enqueue other tasks. These benchmarks represent a set of common modules across the RMS domain. Some of the benchmarks are based on publicly available code, and the remaining ones are based on well-known algorithms. These benchmarks are as follows:

1. The Binomial Tree uses a 1D binomial tree to price a single option. Given a tree of asset prices, the algorithm derives the value of an option at time 0 (that is, now) by starting at time T (that is, the expiration date) and iteratively stepping "backward" toward t=0 in a discrete number of time steps, N. At each time step t, it computes the corresponding value of the option Vi at each node of the tree i. The number of exposed tasks (i.e., nodes) is small at any time step and the task size is small. Hence, task queue overhead must be small to effectively exploit the available parallelism.

2. The Game Physics constraint solver iteratively solves a set of force equations in a game physics constraint solver. For inputs that have few bodies and constraints, the amount of parallelism is limited, especially for a large number of cores.

3. Cholesky, Backward Solve, and Forward Solve are operations on sparse matrices. Cholesky performs Cholesky factorization. Backward Solve and Forward Solve perform backward and forward triangular solve on a sparse matrix, respectively. These solvers use a data structure called elimination tree that encodes the dependency between tasks. The irregularity in sparse matrices results in high variation of task size. Therefore, we need very efficient task management to achieve good load balancing.

4. The Canny Edge Detection computes an edge mask for an image using the Canny edge detection algorithm. It first finds a group of edge candidates and determines whether their neighbors are likely to be edges. If so, the algorithm adds them to the candidate list and recursively checks their neighbors for candidacy. The number of tasks correlates directly to the number of candidates and tends to be small. Further, the amount of work in checking for candidacy is also very small.

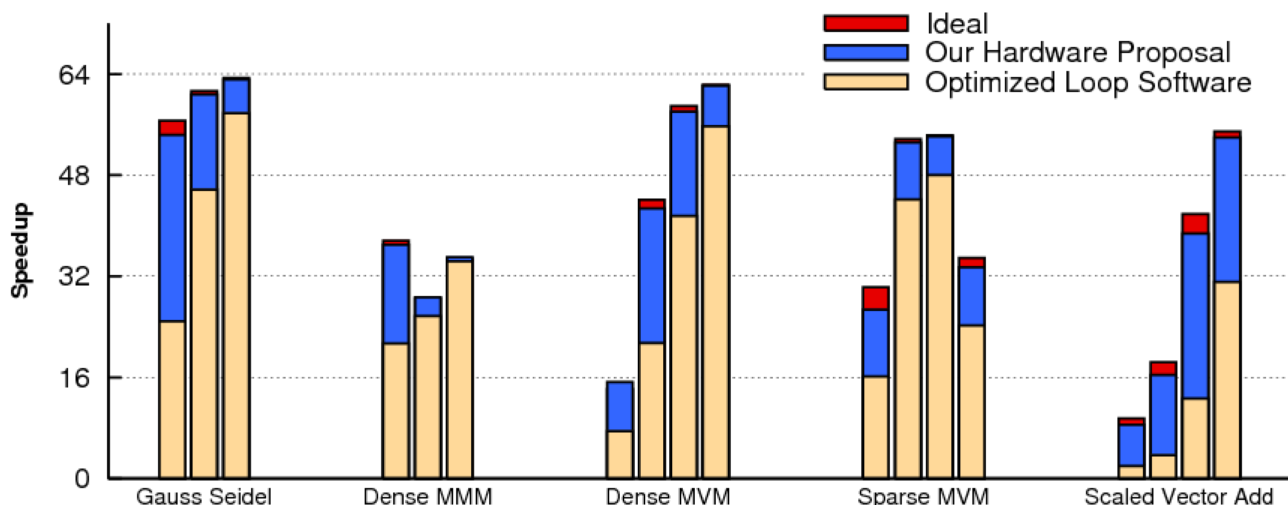

Figure 6: Performance of loop-level benchmarks

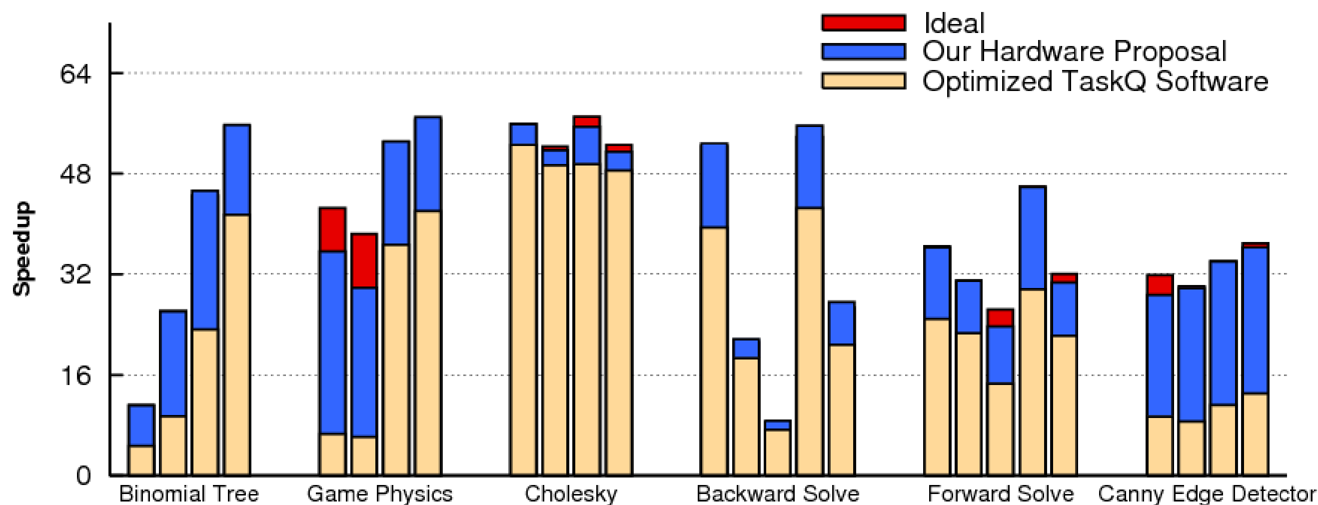

Figure 7: Performance of task-level benchmarks

Results

Figures 6 and 7 show the performance benefit of our proposed hardware for the loop-level and the task-level benchmarks, respectively, when running with 64 cores. In particular, the hardware proposal is compared with the best optimized software implementations and an idealized implementation (Ideal) in which tasks bypass the LTUs and are sent directly to/from GTU with zero interconnect latency. Additionally, the GTU processes these tasks instantly without any latency. The optimized software implementation uses a combination of widely used state of the art techniques [2, 3] that deliver the best performance.

The graphs represent the speedup over the one-thread execution using the Ideal implementation. For each benchmark, they show multiple bars. Each bar corresponds to a different data set shown in Tables 1 and 2.

For the loop-level benchmarks in Figure 6, the proposed hardware executes 88% faster on average than the optimized software implementation and only 3% slower than Ideal.

For the task-level benchmarks in Figure 7, on average the proposed hardware is 98% faster compared to the best software version and is within 2.7% of Ideal. For Game Physics with one data set, and Forward Solve with another data set, the amount of parallelism available is very limited. In the software implementations, the cores contend with each other to grab the few available tasks, which adversely impacts performance.

CONCLUSION

MCAs provide an opportunity to greatly accelerate applications. However, in order to harness the quickly growing compute resources of MCAs, applications must expose their thread-level parallelism to the hardware. We explore one common approach to doing this for large-scale multiprocessor systems: decomposing parallel sections of programs into many tasks, and letting a task scheduler dynamically assign tasks to threads.

Previous work has proposed software implementations of dynamic task schedulers, which we examine in the context of a key emerging application domain, RMS. We find that a significant number of RMS applications achieve poor parallel speedups using software dynamic task scheduling. This is because the overheads of the scheduler are large for some applications.

To enable good parallel scaling even for applications with very small tasks, we propose a hardware scheme to accelerate dynamic task scheduling. It consists of relatively simple hardware and is tolerant to growing on-die latencies; therefore, it is a good solution for scalable MCAs.

We compare the proposed hardware to optimized software task schedulers and to an idealized hardware task scheduler. For the RMS benchmarks we study, our hardware gives large performance benefits over the software schedulers, and it comes very close to the idealized hardware scheduler.

ACKNOWLEDGMENTS

We thank Trista Chen, Jatin Chhugani, Daehyun Kim, Victor Lee, Skip Macy, and Mikhail Smelyanskiy who provided the benchmarks. We thank Pradeep Dubey who encouraged us to look into this problem. We also thank the other members of Intel's Applications Research Lab for numerous discussions and feedback.

REFERENCES

[1] Pradeep Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology@Intel Magazine*, February 2005.

[2] Sanjeev Kumar, Christopher Hughes, Anthony Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *Proceedings of 34th International Symposium on Computer Architecture*, June 2007.

[3] *Intel® Thread Building Blocks Reference*, 2006. Version 1.3.

[4] *OpenMP Application Program Interface*, May 2005. Version 2.5.

AUTHORS' BIOGRAPHIES

Sanjeev Kumar is a Staff Researcher in the Corporate Technology Group. His research interests are parallel architectures, software, and workloads especially in the context of chip-multiprocessors. He received his Ph.D. degree from Princeton University. His e-mail is sanjeev.kumar at intel.com.

Christopher J. Hughes is a Staff Researcher in the Corporate Technology Group. His research interests are emerging workloads and computer architectures, with a current focus on parallel architectures and memory hierarchies. He received his Ph.D. degree from the University of Illinois at Urbana-Champaign. His e-mail is christopher.j.hughes at intel.com.

Anthony D. Nguyen is a Senior Research Scientist in the Corporate Technology Group. His research interests include developing emerging applications for architecture research and designing the next-generation chip-multiprocessor systems. He received his Ph.D. degree from the University of Illinois, Urbana-Champaign. Hise-mail is anthony.d.nguyen at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS® Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from http://www.intel.com.

Additional legal notices at: http://www.intel.com/sites/corporate/tradmarx.htm.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2007 Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information visit: www.intel.com/sites/corporate/tradmarx.htm