Adaptive Scheduling Using Performance Introspection

TR-12-02

2012

Allan Porterfield
akp@renci.org
Renaissance Computing Institute

Rob Fowler
rif@renci.org
Renaissance Computing Institute

Anirban Mandal
anirban@renci.org
Renaissance Computing Institute

David O'Brien
obrien@renci.org
Renaissance Computing Institute

Stephen L. Olivier
olivier @renci.org
Renaissance Computing Institute

Michael Spiegel
mspiegel @renci.org
Renaissance Computing Institute

Adaptive Scheduling Using Performance Introspection

Allan Porterfield, Rob Fowler, Anirban Mandal,
David O'Brien, Stephen L. Olivier, Michael Spiegel
Renaissance Computing Institute (RENCI)
Chapel Hill, NC
{*akp,rjf,anirban,obrien,olivier,mspiegel*}@*renci.org*

Abstract

As energy becomes a driving force in High Performance Computing, determining when and how energy can be saved without impacting performance is a key goal for both HPC hardware and software. Scalability studies have shown that some memory-bound applications do not scale as the thread count increases, and in some cases performance degrades. Adaptive Scheduling *recognizes when an application is in a memory-bound region and throttles the number of active hardware threads. Our RCRdaemon tool acquires hardware performance counter measurements in near-real time. A simple hardware model added to the Qthreads runtime system reads the collected data to determine when memory contention exists. Using that information, our extension to the Qthreads scheduler reduces contention by throttling hardware threads.* Adaptive Scheduling *has very low performance impact both for memory-bound benchmarks (below 4.2%) and for compute-bound benchmarks (2.4% - 3.7%).*

For these techniques to reduce energy costs, additional hardware energy features will be required. Applications using Adaptive Scheduling *can transition from memory-bound to compute-bound regions hundreds of times a second. Hardware mechanisms or instructions to allow energy savings during the short memory-bound regions could be used effectively by multithreaded software to reduce the overall power requirements for memory-bound applications.*

1. Introduction

Moore's law for the number of circuit elements on a high-end chip continues to be applicable, but in recent years it has manifested itself in the form of modular processor chip designs in which it has been relatively easy use die area to add processor cores, cache modules, memory controllers, integrated graphics, and other devices. Current generations of multi-core, multi-socket machines have between 12 and 64 hardware threads and plans for chips with 50+ cores have been announced [1]. In contrast, off-chip communication crosses the perimeter of the die and is less scalable. The impact of Moore's law on memory has been an increase in the number of bits per chip with relatively smaller improvements in latency and bandwidth. To keep the relative cost of memory under control, per core systems are configured with fewer memory sticks than in the past. Sustained high performance on such systems requires simultaneously achieving an adequate degree of parallelism while being constrained by access to shared, off-chip resources with less concurrency than the set of cores.

Power and energy consumption represent both constraints and costs to be controlled in high end systems. Initial studies of exascale systems have estimated total power consumption as high as 154.8 MW [2]. Energy management in large systems is therefore an increasingly important problem.

The software stack of emerging systems must thus deal with a complex environment that will require a high-degree of balanced parallelism that is constrained by shared resources such as memory while using energy efficiently. There is a trend towards more dynamic and adaptive algorithms in HPC. This trend both presents a challenge for systems that need to run them effectively, but it also presents an opportunity to exploit dynamic adaptation to deal with resource constraints while saving energy.

In this paper, we report on experimental results using a prototype runtime system (with compiler support) to throttle parallelism during periods of shared resource contention to maintain or improve performance while exposing the opportunity to reduce energy consumption by idling cores.

A key component of the software stack is a mechanism for measuring the utilization of key shared

resources such as memory and memory controllers. We developed a mechanism called *Resource Centric Reflection* for monitoring and analyzing hardware performance measurements for these resources. Raw counts are converted to utilization figures that have been calibrated experimentally [3], [4]. The analysis is available to other layers of the system.

When a shared resource is saturated, additional offered load does not result in improved performance and may decrease it. By adjusting the offered load to keep utilization just below saturation, performance is not negatively affected and in some cases may improve as resources are shared more effectively by fewer threads (*e.g.*, accommodating a larger cache footprint per core).

An adaptive scheduler uses the utilization models to adjust the workload by varying the number of active cores. The software separates the concept of a software thread from the hardware thread that executes it. The runtime system uses this to maintain a fixed number of software threads interacting in loops and barriers while allowing the actual number of active hardware threads to vary. Furthermore, multi-threaded applications must be compiled to execute correctly in this environment with dynamic hardware adaption.

2. Background

Dynamic adaptive scheduling requires considerable infrastructure. Qthreads is an open-source lightweight threaded runtime that allows the modification of scheduling algorithms. ROSE compiles OpenMP programs for execution using Qthreads. RCRToolkit acquires performance data dynamically with minimal latency and overhead.

2.1. Qthreads

Qthreads is a library supporting portable, high-performance, massive multi-threading [5]. It is loosely modeled on the Tera MTA system [6], which supports many simultaneous lightweight threads in hardware by providing a large number of register sets and interleaving instructions from the various active threads. The Qthreads library instead supports lightweight threads in software by providing compact stacks for each thread and fast context switching. The library is supports IA32, IA64, X86-64, PowerPC, SPARC architectures, and several accelerators.

The software architecture of Qthreads is shown in Figure 1. Each lightweight thread is called a *qthread*. Qthreads are scheduled onto a small set of heavy-weight *worker* threads created using the POSIX threads (pthreads) library [7]. A qthread is the smallest unit of work, such as a set of loop iterations or an OpenMP task, and execution of an application generates many more qthreads than worker pthreads. Each worker pthread is pinned to a processor core and assigned to a group, called a *shepherd*. Multiple worker pthreads can be assigned to each shepherd, enabling the mapping of shepherds to different architectural components, e.g., one shepherd per core, one shepherd per shared L3 cache, or one shepherd per processor socket.

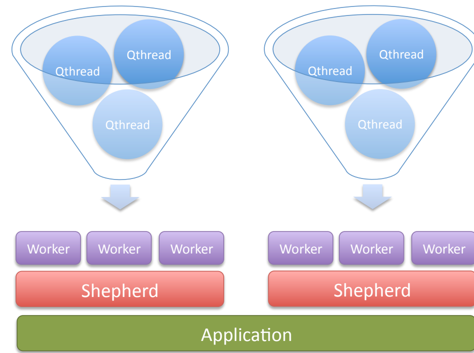


Figure 1: Software architecture of Qthreads.

Like the MTA system, synchronization in Qthreads is supported by full/empty bit (FEB) operations, originally developed for the Heterogeneous Element Processor (HEP) machine [8]. In the FEB scheme, an additional bit is associated with each word in memory. It specifies the current state of the data at that location as either valid (full) or invalid (empty). Each memory access is blocking or non-blocking operation. Blocking operations wait for the value to be in a desired state and set the value to a possibly new state on completion. When qthreads block, e.g., performing a FEB operation, a context switch is initiated. Since this context switch is done in user-space via function calls, requiring neither signals nor the saving of a full set of registers, it is less expensive than an operating system or interrupt-based context switch. This technique allows qthreads to execute uninterrupted until blocked, and, once they have been blocked, allows the scheduler to keep workers busy by switching to other qthreads.

The Qthreads library includes multi-threaded loop execution, built upon the core threading components. The API provides three basic parallel loop behaviors: create a separate qthread for each iteration, divide the iteration space evenly among all shepherds, or use a queue-like structure to distribute sub-ranges of the iteration space to enable self-scheduled loops. These loop configurations are used to support OpenMP loop parallelism. The worker pthreads serve as OpenMP

threads, and qthreads serve as OpenMP tasks.

2.2. OpenMP on Qthreads

Before one can execute OpenMP applications using Qthreads, the application source code must be compiled. Compilation is a two-phase process using both the ROSE source-to-source compiler [9] and the native C++ compiler. ROSE performs syntactic and semantic analysis on the code and transforms the OpenMP directives into function calls in an API called XOMP [10]. XOMP defines a common interface for OpenMP 3.0, abstracting out internal implementation details of the runtime system. Using a runtime library with ROSE is as simple as creating XOMP wrappers for the library, and the ability to hot-swap different implementations of our own runtime system allows fast development and testing. In the second compilation stage, the transformed source code is compiled into executable code by the native C++ compiler and linked with the Qthreads library, which implements the XOMP functions. A benefit of this two-step compilation is ease of porting applications: Since the transformed code is standard C++, it is possible to compile the original source code using ROSE on a development system, then compile the transformed code using the native compiler on another system with a different ISA.

Our ROSE infrastructure generates guided-self scheduled rather than static scheduled parallel loops. This is important for *Adaptive Scheduling*, because it uses the over-partitioning of work to guarantee load balancing.

ROSE and the XOMP interface provide a means of running applications on Qthreads (and our schedulers) without rewriting the application to use a new runtime interface. This greatly simplifies the comparisons between the various runtime implementations.

2.3. Hierarchical Scheduling

The Qthreads multi-threaded shepherds task scheduler [11] uses a hierarchical approach combining the low-overhead load balancing of work stealing schedulers [12] and the shared cache exploitation of parallel depth-first schedulers [13]. One shepherd is created for each chip/socket. On the systems being studied, the cores on a chip all share a single L3 cache and all are proximal to a local memory attached to that socket. Within each shepherd, one worker is mapped to each core. Among workers in each shepherd, a shared LIFO queue provides depth-first scheduling close to serial order to exploit the shared cache. Thus, load-balancing happens naturally among the workers on a chip and concurrent tasks have possible overlapping localities that can be captured in the shared cache.

Between shepherds, work stealing is used to maintain load balance. Each time the shepherd's task queue becomes empty, only the first worker to find the queue empty goes to look for more work. It steals enough tasks (if available) from another shepherd's queue to supply all the workers in its shepherd with work. The other workers in the shepherd spin until the stolen work appears. Aggregate task queuing for workers within each shepherd reduces the need for remote stealing and decreases the number of probes required to find available work by a factor of the number of workers per shepherd. While a shared queue can be a performance bottleneck, the number of cores per chip is bounded, and locking operations are fast within a chip.

Hierarchical scheduling separates workers by shepherd, one shepherd per processor chip for most current systems. *Adaptive Scheduling* uses this structure during throttling to reduce threads evenly across the system.

2.4. Measuring Resource Utilization

Measuring the utilization of shared resources is a key requirement for *Adaptive Scheduling*. Recent generations of high-end processor chips have integral shared caches, memory controllers, and inter-chip communication. These resources are independent, loosely synchronized state machines that together comprise the *uncore* (in Intel's terminology). They have their own performance measurement sensors and counters. For several reasons, using this hardware to provide introspective monitoring of chip- and node-wide behavior required the development of our own tool set, called RCRToolkit. One problem is that on Intel systems the Linux PerfEvents facility does not provide access to these counters. On AMD processors, uncore events are mapped to on-core counters, so PerfEvent access is possible if done very carefully using a core not used by the application. A second issue is that research performance tools layered on top of PerfEvents such as PAPI [14], TAU [15], and HPCToolkit [16] provide *first person* views of performance that are designed to attribute performance problems in a single thread to specific sections of code. Counters and events are virtualized, bound to a specific thread, and advance only when that thread is running. If multiple threads attempt to monitor a specific event, there is either a conflict/error or all of the concurrently executing threads see overlapping sets of event instances. In these cases, results are either unobtainable or not usable.

In contrast, the performance measures of interest to us are a consequence of the collective behavior

of all threads running on the box. To address these issues, we developed RCRToolkit to provide chip- and node-wide third-person views of the behavior of shared resources. RCRToolkit has three main parts: RCRdaemon collects and records dynamic hardware counter values and rates; RCRLogger creates logs that combine performance and execution context information; RCRTool's GUI post-processes and displays the data. RCRLogger and RCRTool together provide an effective interface for human monitoring of executions and for off-line analysis. RCRdaemon also makes its information available to other system software via a shared memory segment organzied as a blackboard with several single-writer, multiple-reader sections to permit lock-free, low overhead communication. Applications, the thread scheduler, and other software can make notations regarding execution context on the blackboard for RCRLogger. In section 4.2.2, screen-dumps from the RCRTool GUI are shown to explain observed performance.

RCRdaemon runs as root and is pinned to a specific "management" core that can be kept separate from the pool of cores available for application threads. On AMD processors Linux PerfEvents can be used, but on Intel systems the model specific register (MSR) kernel module is used directly to access the off-core counters. In both cases, libpfm supports symbolic event names. At startup, RCRdaemon reads a configuration file that specifies the events to monitor, the sampling rates, some simple models to transform raw counts into measures of concurrent access at each resource, and some thresholds on the levels of concurrency that define "overloaded" and "idle" state. There is a unique configuration for each model of compute node. In typical operation on a reserved core of a 2.00 GHz Intel Xeon X7550, the configuration polls the counters updated between 6000 and 7000 times per a second. At those rates, RCRdaemon uses between 12% and 17% of a single core, or 0.5% of the 4-socket 32-core node. Overheads can be reduced by lowering the sample rate.

3. Implementation

Dynamically adjusting computation to react to changes in system state requires assembling the components discussed above into a single cohesive scheduling package. Data must be acquired in near real-time, the runtime must allow hardware threads to enter and leave the computation, and a performance model must decide when contention exists.

3.1. Adaptive Scheduling

Adaptive Scheduling uses information about the current state of the system to decide the correct number of hardware threads running at any given time. *Adaptive Scheduling* receives input not only from the application, in the form of a queue of threads to execute, but also from RCRdaemon, in the form of utilization rates of system resources. This additional set of inputs allows application execution to be tuned to the current conditions. Initially, the only control available for tuning is to change the number of active hardware threads.

Reducing the number of active threads to improve system performance seems counterintuitive, but is useful in at least two situations. When resources do not overload gracefully, then *Adaptive Scheduling* provides the runtime with the ability to prevent thrashing. For example, if a thread uses 1/3 of the available cache and has billions of cache hits and only start-up misses, one thread running in isolation performs well. However, executions with four or six threads exceed the cache capacity, and almost every reference goes to main memory. Limiting the active threads to three would maximize cache performance. The second situation is when factors other than time-to-completion are critical to overall system performance. In the near term, energy will be such a factor for massive systems like the exascale systems being discussed for deployment near 2020 [2]. Current power estimates for Exascale systems above 150 MW and may even be a deal-breaker. Energy is also a critical resource for any mobile device. If the runtime can identify portions of applications in which the threads can be either slowed or stopped, the possibility exists for significant energy savings. While not a total solution, it is a plausible part of the package of solutions that will be needed. For mobile devices, the runtime can be even more aggressive, trading performance for battery life.

The implementation of *Adaptive Scheduling* combines Qthread's Hierarchical Scheduling with the information available from RCRdaemon. The RCRdaemon is always running, evaluating the counters. It follows user-specified thresholds to flag each rate as saturated, normal, or idle. During program initialization, within the runtime, a thread is started that enters a loop to read the current state information from RCRdaemon and calculate the current "correct" number of active threads. When each hardware thread looks for work to execute, either another task or more iterations of the current parallel loop, the "correct" number of threads is checked. If the count of active hardware threads exceeds that number, the hardware thread is put to

sleep. If the number of correct threads goes up, the parallel loop completes, or the program terminates, the sleeping threads are released to find new work or stop.

The key to making *Adaptive Scheduling* work in Qthreads is not the scheduler, but the implementations of barriers and synchronization. If barriers required the arrival of all hardware threads, then "sleeping" a thread would cause significant delays while the system waited for contention to clear before allowing the thread to continue and reach the barrier. In Qthreads, barriers are performed on software threads. Upon reaching the barrier with its current software thread, a hardware thread can context-switch to a new software thread and continue execution. Barriers are still fast, allowing for better performance when system loads are not balanced (OS jitter), and enabling thread-sleeping within *Adaptive Scheduling*. Synchronization is handled with the same mechanism. If a thread needs a value and it is not available, the hardware thread swaps out its software thread and picks up one that is ready to execute.

In the long-term, the benefits of *Adaptive Scheduling* are greatly hampered by the lack of ways to "sleep" a core that save energy and still allows fast wake up. To activate and wake up Dynamic Voltage and Frequency Scaling (DVFS) currently takes milliseconds, but the system conditions can change in microseconds or even nanoseconds. Currently, Qthreads puts the thread into a spin loop. The frequent instruction execution of the spin loop greatly limits potential energy savings. Hardware support to "sleep" a hardware thread that saved energy and could awaken in a few microseconds (or faster) would unlock the potential energy savings of *Adaptive Scheduling*.

The performance model used by *Adaptive Scheduling* is broken into two pieces. The first piece, inside RCRdaemon, determines whether any of the current utilization rates indicate resource overload. The second piece, inside the Qthread runtime system, determines the correct number of threads to have active. On start-up RCRdaemon reads a configuration file that specifies the parameters of its part of the model for that execution. Inside that file, the various rate triggers are easily changeable but transparent to the application end user, who only wants the application to complete quickly. Inside Qthreads, the various rates fire triggers that activate a simple fixed model either to reduce or to increase the number of threads. For the Intel 7750 processor used in the evaluation, the model reduces hardware threads by 1/4 when contention is detected and resets the number of active hardware threads to the full processor width when contention subsides.

Both models are simple and are used to test the principles and basic mechanisms. In future, we expect to add more complex models to both Qthreads and RCRdaemon. Tuning the models to specific processor families in the future may allow further improvements in energy and performance.

3.2. Memory Concurrency

The number of cores on a processor chip and the capacity of DIMMs are both increasing at a faster rate than memory access times, despite improvements in pin speed and caching within the DIMMs. Thus, more and more applications are becoming memory bound. To explore this effect, our initial implementation of the *Adaptive Scheduling* framework focuses on memory concurrency. There is a limit to the number of concurrent memory references a DIMM can handle, exceeding that limit results does not increase memory bandwidth only increase memory latency.

Tuning the scheduler and calculating the new performance rates is part of the porting process to any new processor family. On most systems including our Intel 7750, there is no one single performance counter that measures memory bandwidth or contention within the memory controller. This value can be computed using the "uncore" B-Box counter event IMT_VALID_OCCUPANCY to track the average occupancy of the In-Memory Table. Accessing the "uncore" counters on the Intel is not currently supported by Linux PerfEvents, so access to the required counters is obtained using the MSR interfaces directly. MSR reads and writes are a very low level interface and requires RCRdaemon to run at system protection level.

4. Evaluation

To perform an initial evaluation of the effectiveness and overheads of *Adaptive Scheduling*, we ran a set of OpenMP task benchmarks and two OpenMP benchmark applications known to be memory-bound. This set allows examination of overheads both when throttling occurs and when it does not.

The test system for our experiments is a Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core Nehalem-EX processors installed for a total of 32 cores. Each processor has an 18MB shared L3 cache. It runs CentOS Linux with a 2.6.35 kernel. Although the x7550 processor supports HyperThreading (Intel's simultaneous multithreading technology), we pinned only one thread to each physical core for our experiments.

4.1. Barcelona OpenMP Tasks Suite

Adaptive Scheduling was evaluated with benchmark

	Qthreads	with RCRdaemon	RCR Difference	with Adaptive Scheduling	Adaptive Difference	# Time Invoked
Alignment-**for**	1.025	1.051	-2.5%	1.047	-2.1%	0
Alignment-**single**	1.031	1.061	-3.7%	1.066	-3.7%	0
Nqueens	1.616	1.660	-2.7%	1.666	3.0%	0
SparseLU-**single**	4.548	4.662	-2.5%	4.662	-2.5%	0
SparseLU-**for**	4.535	4.646	-2.4%	4.645	-2.4%	0
Health	1.111	1.066	4.0%	1.058	4.7%	1
Sort	1.089	1.095	-0.5%	1.155	-3.9%	3
Strassen	10.708	10.701	0.06%	11.161	-4.2%	65

Table 1: BOTS Execution Times

	Adaptive Time	# Times Thread Count Reduced	Perf. Change	Idle Time	Ave. Time Idle	Increased Exec. Time
Health	1.058	1	4.7%	7.352	0.229	-0.043
Sort	1.155	3	-3.9%	1.779	0.055	0.066
Strassen	11.161	65	-4.2%	34.496	1.078	0.453

Table 2: Baseline Adaptive Scheduling Statistics

applications from the Barcelona OpenMP Tasks Suite (BOTS), version 1.1, The suite comprises a set of task-parallel applications from various domains with varying computational characteristics [17]. The following benchmark components and inputs were used:

- *Alignment*: Aligns sequences of proteins using dynamic programming (100 sequences)
- *Health*: Simulates a national health care system over a series of timesteps (144 cities)
- *NQueens*: Finds solutions of the n-queens problem using backtrack search ($n = 14$)
- *Sort*: Sorts a vector using parallel mergesort with sequential quicksort and insertion sort (128M integers)
- *SparseLU*: Computes the LU factorization of a sparse matrix (10000×10000 matrix, 100×100 submatrix blocks)
- *Strassen*: Computes a dense matrix multiply using Strassen's method (8192 x 8192 matrix)

Two of the tests, *Alignment* and *SparseLU*, have multiple versions: one that uses a **for** loop to generate the parallel tasks, and a second that has a **single** parent task create the parallel tasks.

Table 1 shows the best result from ten trials of each BOTS test in several circumstances. The standard deviations were small. For the first set, we used the default Qthreads scheduler with 32 threads. For the second set, labeled "with RCRDaemon", we enabled monitoring of the activity via the RCRdaemon but not throttling of threads. For the third set, labeled "with Adaptive Scheduling", we enabled throttling.

No memory-bound regions were detected in *NQueens*, *Alignment*, or *SparseLU*. The thread count was never limited, and performance degradation ranged from 2.5% to 3.7%, in line with expectations. The overhead of enabling *Adaptive Scheduling* is approximately the cost of one core in compute-bound benchmarks. As core counts rise this overhead will decrease.

Table 2 shows the performance of *Health*, *Sort* and *Strassen* with memory contention in more detail.

4.1.1. Health. Performance on the *Health* benchmark improves with fewer threads. We speculate that using fewer threads provides better load balance and results in less contention for memory resources. The program is memory-bound for the entire execution. *Adaptive Scheduling* reduces the thread count early in execution and never raises the total until the program is completed. During the 1.058 second execution time (the best time observed), 1/4 of the threads were idle for 1.047 seconds or 98.9%.

The benchmark fastest overall performance was with with 1/4 threads idle for nearly all of the application's execution. Reducting thread count halved the number of long latency (1024+ cycles) loads. The reduction in long latency references may be the result of fewer threads of execution producing a stream of references that hit on DIMM cached pages with much greater frequency, requiring fewer DIMM page reloads.

Since threads are idle for long periods, *Adaptive Scheduling* could achieve energy savings with this application even with today's high-latency per-core power adjustment controls.

4.1.2. Sort, Strassen. *Sort* idled threads 3 times during execution. Each thread was idle for about 1/20th of the total execution. At any time at most 1/4 of threads

	Baseline Adaptive	Lazy Adaptive	Diff	# Times Reducded	Idle Time	Aggr Adaptive	Diff	# Times Reduced	Idle Time
Health	1.058	1.052	4.8%	1	7.316	1.073	3.0%	1	7.419
Sort	1.155	1.224	-12.2%	1	8.125	1.217	-11.6%	1	8.298
Strassen	11.161	12.334	-15.3%	8	68.790	11.361	-6.2%	225	24.424

Table 3: Comparison of Throttling Strategies

	Thread count							
	4	8	12	16	20	24	28	32
Qthreads	10.30	5.28	3.82	3.39	3.21	3.16	3.11	3.09
Adaptive	13.41	5.97	4.46	3.70	3.52	3.32	3.20	3.13

Table 4: *Heat* Execution times (best of 10)

are idled, corresponding to throttling set about 20% of the time. The three throttling episodes results in a slowdown approximating the amount of time spent in an idle state. For relatively performance-neutral programs, the overall cost of throttling is heavily impacted by the cost of entering and leaving any power saving idle modes. The total overhead costs during execution are dominated by this transition cost.

Running *Strassen* with *Adaptive Scheduling* results in a 4.2% slowdown. Each thread is idle for just under 10% of the execution time. It was throttled almost 6 times every second during the 11+ seconds of execution. *Strassen* spent nearly half of the time throttling execution, but no single throttling event lasted for an extended period. Power saving opportunities were short and would require lightweight per-core power adjustment support.

4.1.3. Throttling Strategies. Parameters to the *Adaptive Scheduling* model control how quickly it responds to the onset of resource contention and how quickly it responds once the contention is over. We tested three versions: the "Baseline" version, used in the previous experiments, a "Lazy" version that is slow to recognize contention and idle, and an "Aggressive" version that determines contention and idle states very quickly.

Table 3 shows the performance the two variations on the three memory-bound tests. On *Health*, Baseline and Lazy performed nearly identically with only 1 throttling event lasting almost all of execution. Aggressive also had 1 event, but it occurred earlier in the execution and total execution was slightly longer (1.4%). Lazy on *Sort* and *Strassen* produced substantially fewer transitions into and out of throttling. Both are throttled for a much larger percentage of execution, but this results in execution slowdowns above 12%. This shows the importance of being able to use short throttling sequences to reduce power consumption.

Aggressive, however, demonstrates that switching too fast and too often does not necessarily perform better. Execution time is higher than for Baseline and for *Strassen* the amount of time in idle actually drops. *Adaptive Scheduling* has the opportunity to save power during the execution of memory-bound applications, but the effectiveness will be determined by how fast the system can recognize contention and how fast the system can recover when the contention is removed.

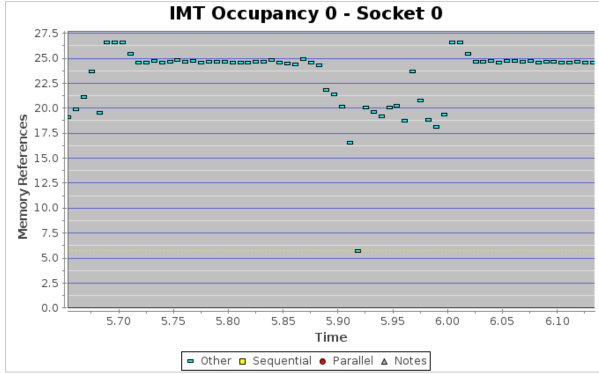
4.2. Memory-Bound Tests

In addition to BOTS, we tested two memory-bound applications *Heat* and NAS *IS* with *Adaptive Scheduling* to better understand the execution effects of dynamically throttling parallelism.

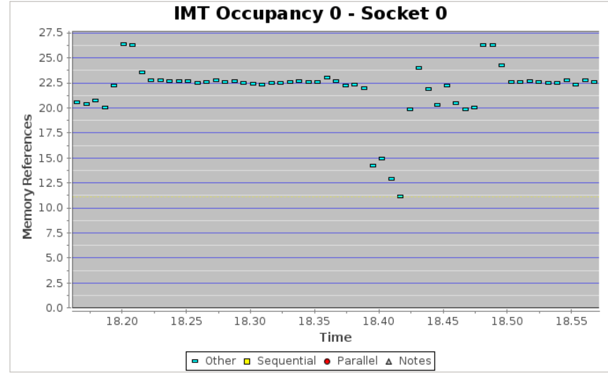
4.2.1. Heat. *Heat* is a 2D heat diffusion simulation from the example set in the MIT Cilk distribution [18] that we ported to OpenMP 3.0. It is a small application known to be effectively memory-bound for large portions of its execution.

During the 3.13 seconds of execution, *Adaptive Scheduling* reduced the number of threads 272 separate times. On average each thread was idle for 0.49 sec (total idle = 15.20 seconds) or 15.6%. Since at most 1/4 of threads are idle at any one time, threads were limited for greater than 60% of the total execution. Although the threads were limited, the performance degradation was only 1.1%. Phases of Heat vary in memory intensity, and dynamic tracking of hardware performance allows substantial periods of core idling with minimal impact on application performance.

4.2.2. NAS IS. The memory-bound behavior of the *IS* benchmark from the NAS Parallel Benchmark suite suggested it as a good candidate for *Adaptive Scheduling*. Our evaluation used the Class 'C' problem size. Reported results are the best of ten trials for each



(a): IS 32 threads



(b): IS 24 threads

Figure 2: RCRTool Graphs for one iteration of NAS *IS*

	Thread count						
	8	12	16	20	24	28	32
static (GNU OMP)	3.68	3.01	2.77	2.70	2.77	2.91	3.05
static (Qthreads)	3.54	2.87	2.65	2.59	2.60	2.71	3.05
guided (Qthreads)	3.51	2.88	2.69	2.68	2.80	2.96	3.14
guided (Adaptive)	3.98	3.85	3.29	3.43	3.18	3.24	3.35

Table 5: NAS *IS* Execution times (best of 10)

configuration, and for all tests the standard deviations of the 10 runs was small.

The GNU OpenMP implementation using static scheduling of loop iterations, shown in Table 5, results in execution times rising from 2.70 to 3.05 (12.9%) as the hardware thread counts rises 20 to 32. Switching to Qthreads increased the performance of lower thread counts but the performance of 32 threads was constant for a greater performance degradation of 17.7% If the thread count is reduced automatically to 24 (6 per core), then we expected that falloff would be eliminated. In practice, two problems were discovered: explicit static scheduling of the loop iterations and the initial thread count determining the amount of work.

Static Scheduling. The original implementation of *IS* uses static scheduling of the loop iterations to cores. This exploits memory locality, maintains load balance, and allows compiler optimization of threading overheads. However, it also prevents thread assignment to cores from being modified during execution, effectively preventing *Adaptive Scheduling*. Iterations are assigned to the threads before the loop starts, and they never need to enter the runtime to get more loop iterations. As implemented, there is no hook for *Adaptive Scheduling* to reduce the active worker count. Even if the hook existed, the requirement that the threads execute on specific cores would greatly reduce the potential for dynamic improvement. Threads would halt and wait for the dynamic adaptation to recognize an idle state before completing the loop. This overhead would enevitably be high and starvation could easily occur if threads do work outside the loop.

We changed the loop iteration scheduling of *IS* to guided self-scheduling. This resulted in a maximum 3.5% increase in execution time, a result of additional overhead acquiring loop iterations. The additional overheads were only partially offset by better load balance for all but the lowest thread counts.

Increased Work. Even with the change to guided self-scheduling of loop iterations, *Adaptive Scheduling* was 6.6% slower than normal Qthreads. To understand this performance degradation required a more detailed study of the *IS* implementation. The algorithm performs 10 iterations of three loops. The first histograms the array multiple buckets (one per thread to eliminate data races), the second compresses the buckets to a single bucket, and the third assigns the data to the correct location. The difficultly is that the work in the second step is proportional to the number of threads *potentially* doing the work. Even when the thread count is reduced, the extra work is done (by fewer threads) and the overall execution time increased.

Figure 2 shows RCRTool snapshots of a 32 thread execution and a 24 thread execution. RCRTool allows the user to examine information about each point. Although 32 threads has slightly higher memory con-

currency (24.5 vs 23), looking at the first and last point of flat memory usage (the histogram step), both graphs have the same number of points (24) and execution times of about 0.165 seconds.

Measuring to the beginning of the next loop (compress and assign steps), 24 threads only takes .115 seconds while 32 threads takes .144 secs. Each array to be compressed takes about .004 seconds (assuming several of the points correspond to the assignment of the correct locations). For this size problem, the system has almost as much work assembling the parallel pieces as actual sorting.

The net effect of these two overheads for *IS* is that although 1/4 of threads can be disabled by adaptive scheduling for the duration of the execution. Throttling occurs 6 times during execution for about 83% of the total execution time and each thread was idled for about 0.931 seconds.

5. Related Work

In a previous study [19] we used a method similar to working set scheduling [20] to combine cache and TLB miss rates with throughput measurements to throttle the number of concurrent queries in the relational database system of a three-layer architecture running a transaction processing benchmark. Without throttling, system throughput exhibited a classic "mortar shot" curve in which throughput decreased with increasing load. Throttling stabilized this at near peak performance.

Several research efforts have focused on mitigating memory contention for concurrently executing applications with independent address spaces. Monitoring systems have been introduced at the operating system level to provide hints to the OS scheduler when memory contention is detected [21], [22], [23], [24]. These schedulers attempt to schedule memory-bound and cpu-bound processes together. Compiler-transformation techniques have also been proposed as a means to mitigate process contention for application quality of service [25], [26].

Other research activities have focused on reducing memory contention for concurrently executing threads with shared address spaces. Prior works have focused on data parallel applications, disabling threads in parallel loops in the presence of contention [27], [28]. Our contention management strategy has been designed for both task-parallel and data-parallel OpenMP programs.

Another strategy for reducing memory contention uses replacement libraries for thread creation, synchronization, and communication to renegotiate the mapping of user-level threads to operating system (OS) resources. The replacement libraries can either map several user-level threads to a single OS thread [29] or map user-level threads to OS processes [30]. These techniques are library-specific, in that they may produce unsafe transformations for programs that use synchronization primitives external to libraries that have been replaced. A third technique is to use a cache simulation to model the effects of cache contention, typically at a performance expense relative to executing native code [31], [32].

6. Future Work and Conclusions

We expect more memory-bound applications in the future, because of trends in both computer architecture (reduced memory bandwidth per core) and applications (dynamic adaptive and sparse). Energy consumption is also becoming a major issue for both Exascale computation and mobile devices. *Adaptive Scheduling* uses performance introspection to adjust application demands to current system conditions. The overhead cost is low, and we expect it to fall as core counts rise. Reduced contention in some cases improves performance with better load balance and/or improved cache utilization. More commonly the impact on performance is minimal. Throttling provides the software the ability to dynamically reduce power consumption.

To effectively use techniques like throttling and *Adaptive Scheduling* to reduce power consumption, new hardware features must be available to the runtime. The runtime reacts to system load, but it does not have knowledge of how long contention will continue. To effectively save energy, the runtime will require a hardware mechanism for core power reduction that can be reversed in 10's or low 100's of nanoseconds without OS intervention. Such a mechanism could be effectively used in a many-core system to save power without substantially impacting performance.

Adaptive Scheduling is early work using performance introspection to improve application performance. Work will continue to find better dynamic models to determine times to transition into and out of throttled execution. Shared resources beside memory concurrency will be examined, including both hardware, e.g., shared cache performance and network load, and software resources, e.g., file I/O. Our interests also include better methods to relate execution performance to the application tuner, which will include understanding when throttling occurs and its effects.

7. Acknowledgments

This work is supported in part by a grant from the United States Department of Defense.

References

[1] "Intel MIC news release." [Online]. Available: http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm

[2] P. Kogge, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] A. Mandel, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, White Plains, New York USA, March 2010.

[4] A. Mandal, M. Lim, A. Porterfield, and R. Fowler, "Implications for applications and compilers of multi-core memory concurrency," in *Poster at International Workshop on Languages and Compilers for Parallel Computing (LCPC'10)*, 2010, (poster).

[5] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IPDPS 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[6] G. A. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. J. Smith, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in *Proceedings of the 6th international conference on Supercomputing (ICS '92)*, 1992, pp. 188–197.

[7] IEEE, "Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]," IEEE, IEEE Standard 1003.1c–1995, 1995.

[8] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *4th Symposium on Real-Time Signal Processing*. SPIE, 1981, pp. 241–248.

[9] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel processing letters*, pp. 215–226, 2000.

[10] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP*, ser. LNCS, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 15–28.

[11] S. Olivier, A. Porterfield, K. Wheeler, and J. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *International Workshop on Runtime and Operating Systems for Supercomputers*, Tuson, AZ, USA, June 2011.

[12] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, pp. 720–748, September 1999.

[13] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM*, vol. 46, no. 2, pp. 281–321, 1999.

[14] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000.

[15] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 287–311, May 2006.

[16] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 685–701, April 2010.

[17] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *ICPP '09: Proc. 38th Intl. Conference on Parallel Processing*. IEEE, Sept. 2009, pp. 124–131.

[18] "The Cilk project, release 5.4.6." [Online]. Available: http://supertech.csail.mit.edu/cilk/

[19] R. Fowler, A. Cox, S. Elnikety, and W. Zwaenepoel, "Using performance reflection in systems software," in *Proceedings of USENIX Workshop on Hot Topics in Operating Systems (HOTOS IX)*, Lihue, HI, Mar. 2003, extended abstract.

[20] P. J. Denning, "The working set model for programming behavior," *CACM*, vol. 11, no. 5, 1968.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, pp. 54–66, May 2008.

[22] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *21st Euromicro Conference on Real-Time Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 194–204.

[23] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 47–58.

[24] M. Ghosh, R. Nathuji, M. Lee, K. Schwan, and H.-H. S. Lee, "Symbiotic scheduling for shared caches in multi-core systems using memory footprint signature," in *International Conference on Parallel Processing (ICPP 2011)*, 2011, pp. 11–20.

[25] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 257–265.

[26] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for QoS in warehouse scale computers," in *Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2012.

[27] C. Jung, D. Lim, J. Lee, and S. Han, "Adaptive execution techniques for SMT multiprocessor architectures," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 236–246.

[28] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 277–286.

[29] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 270–279.

[30] T. Liu and E. D. Berger, "SHERIFF: precise detection and automatic mitigation of false sharing," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 3–18.

[31] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 27–38.

[32] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the 4th annual workshop on modeling, benchmarking, and simulation*, 2008, pp. 28–36.