IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Multiple Multithread Applications on Asymmetric and Symmetric Chip MultiProcessors

Tomer Y. Morad, Avinoam Kolodny, Uri C. Weiser

CCIT Report #701
August 2008

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #701 August 2008

Multiple Multithreaded Applications on Asymmetric and Symmetric Chip MultiProcessors

Tomer Y. Morad† Avinoam Kolodny† Uri C. Weiser†

†Department of Electrical Engineering
Technion, Haifa, Israel
{tomerm@tx, kolodny@ee, uri.weiser@ee}.technion.ac.il

Abstract

This paper evaluates new techniques to improve performance and efficiency of Chip MultiProcessors (CMP) for workloads consisting of multiple multithreaded applications. Multithreaded applications contain serial phases (single thread) and parallel phases (many threads). While scheduling threads, current techniques do not differentiate between these two phases, resulting in sub-optimal usage of the multiprocessor resources. In this paper, we propose a new thread scheduling mechanism which takes into account the different requirements of each phase, granting higher priority to applications during their critical-serial phases. The advantages of the proposed scheduling mechanism, shown by analytical and experimental evaluation, are threefold. First, system throughput and power efficiency are improved by making better use of the available multiprocessor computing power. Some of the benchmarks show system throughput improvements of as much as 16%. Second, fairness in resource allocation between the applications is improved by as much as 26%. Third, the jitter in execution runtimes in different runs of the same set of applications is reduced by up to 88%. The analysis is performed for asymmetric multiprocessors, where some of the computing cores are faster than others, as well as for symmetric multiprocessors in which all cores are identical. All experiments in this paper are performed in a real environment, consisting of full benchmarks running on a real multiprocessor and operating system.

1 Introduction

Multithreaded applications can take advantage of the added computing ability offered by today's multiprocessors by executing in parallel on many cores. With an ever-increasing core population embedded in state-of-the-art systems [15], the use of multithreading in applications is expected to increase. In this paper, we strive to improve system performance as measured by several metrics when scheduling multiple multithreaded applications in parallel on asymmetric multiprocessors (where some computing cores are faster than others), as well as on symmetric multiprocessors (where all cores are identical).

When examining multithreaded applications, one can identify two types of execution phases, serial phases and parallel phases. In serial phases only one thread is active, whereas parallel phases are comprised of many concurrently active threads. Typically, data preparation for the parallel phases and inherently sequential calculations are done in the serial phases. The heavy independent calculations are performed in the parallel phases.

When two multithreaded applications are run simultaneously, the serial thread of one application may be available for execution together with the parallel threads of the other application. Fig. 1 shows an example of the four possible joint states of two multithreaded applications. The vertical axis represents time, advancing from top to bottom. At each point in time, the number of active threads for each application is shown.

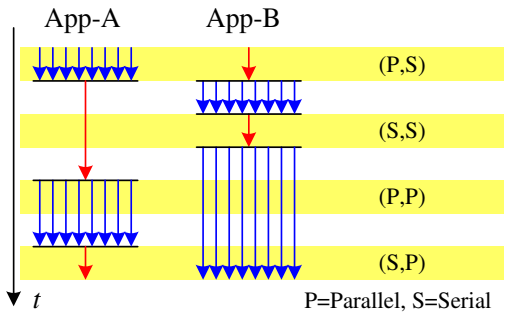

Fig. 1. Illustration of joint states of two sample applications running simultaneously.

Current operating system schedulers, such as the Linux scheduler [1], are not aware of the phases of the running applications. When multiple multithreaded applications are run in parallel, this lack of awareness results in lower throughput, jitter in applications' runtimes, and unfairness between applications. These undesired characteristics may happen because the serial phases, which are critical bottlenecks for the applications, compete for CPU time with the many concurrently executing parallel threads. If these serial phases were executed quickly, the application's bottlenecks would be freed, allowing the application to take advantage of the multiprocessor resources by using many threads.

In this paper, we propose to add another dimension to the current scheduling policy by using information about the parallel and serial phases of the applications. Our proposed scheduler monitors the number of active threads in each application, and hence it can identify and grant higher priority to serial threads.

Fig. 2 shows the four possible joint-states of two applications executing in parallel: (Serial, Serial: S,S), (Serial, Parallel: S,P), (Parallel, Serial: P,S), and (Parallel, Parallel: P,P), as well as the possible transitions among them. The large arrows on state transition arcs denote the most likely transition. The proposed scheduler, shown in Fig. 2 (b), favors the serial thread, thus increasing the probability for transition from (S,P) and (P,S) states to (P,P) state. Current OS schedulers, however, treat the serial and parallel threads equally, thereby lengthening the time required for the serial application to transition into its parallel phase. Current OS schedulers therefore favor the transition from (S,P) and (P,S) to (S,S) state. As a result of the reduced time spent in (S,S), which is the only state in which the multiprocessor has idle cores, the proposed scheduler is expected to improve throughput.

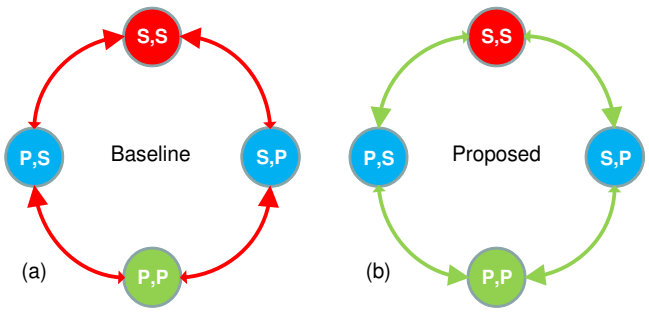

Fig. 2. Illustration of the four possible joint states of two applications running in parallel.

Similarly to the symmetric case, when two multithreaded programs run on asymmetric multiprocessors, serial phases should be favored over parallel phases by being assigned to run on the *faster cores*.

In this paper, we propose a scheduler that grants higher priority to applications in their serial phases in order to increase the multiprocessor throughput, improve fairness and reduce the jitter in execution runtimes. The expected improvements are quantified by a simple analytical model. We validate our proposed techniques by experiments running on a real symmetric CMP with a current version of the Linux operating system, with multiple multithreaded applications executing in parallel. We also validate our techniques on asymmetric structures that are emulated on the real symmetric CMP, with the addition that serial threads are granted higher priority to run on faster cores.

2 Related Work

There are various papers addressing scheduling of single-threaded applications on asymmetric/heterogeneous multiprocessors, which are based on sampling of runtime performance on the different core types. Kumar et al. [11] have proposed a scheduler for multiple single-threaded applications on a heterogeneous multiprocessor. Bower et al. [6] have shown the impact on thread scheduling in symmetric multiprocessors that become heterogeneous during runtime due to frequency scaling, process variations and physical faults. Winter et al. [20] explored thread assignment algorithms for single-thread applications on such multiprocessors.

Other papers address the scheduling problem for a single multithreaded application running on an asymmetric multiprocessor [3][5][10][13]. Grochowski et al. [3][10] have proposed a static scheduling mechanism, implemented at the application level, which schedules the serial phases of applications on the high performance core. They have shown significant performance improvements over symmetric designs with the same power consumption. Balakrishnan et al. [5] proposed a dynamic scheduler for a single multithreaded application on a heterogeneous multiprocessor. They have shown that by scheduling the serial phases on the high performance core, performance increases and the jitter in runtimes of different executions is reduced. We extend these methods [3][5][10] for multiple multithreaded programs, while addressing the

scheduling problem that arises when there are more threads than cores in the multiprocessor.

Fedorova et al. [8] proposed a scheduling algorithm that reduces completion time jitter of single threaded applications on heterogeneous multiprocessors. This was done by granting each thread equal CPU time on each core of the heterogeneous multiprocessor, at the expense of many thread migrations. Their algorithm does not, however, take advantage of the asymmetric structure, which allows the acceleration of critical path threads.

Many papers explore fairness and throughput in SMT architectures [9][12][17][18]. We use and extend their throughput and fairness metrics for asymmetric multiprocessors. Mutlu et al. [14] have shown that fair allocation of memory accesses in DRAM controllers can have a big impact on overall throughput and fairness between applications. Chen et al. [7] have shown that thread assignment in multiprocessors should take into account cache sharing effects between threads. The concepts we present in this paper may co-exist synergistically with these methods.

3 Emulation Environment

All measurements in this paper are performed on an 8-core multiprocessor consisting of four dual-core 2.66MHz XEON processors with 8GB of memory, and with SMT disabled. The operating system used is Linux 2.6.18, and is referred to in this paper as the baseline scheduler. Our benchmarks include the entire SPEC-OMP2001 [4] suite with the medium reference input sets, with the exception of "galgel" because of compilation difficulties in our setup.

OpenMP offers various scheduling options for its parallel constructs [16]. We altered the default OpenMP scheduling policy from static, in which each thread receives an identical portion of the workload, to dynamic, in which each thread consumes a predefined small subset of the workload, and then requests additional work. This is similar to what was done in [5] and [13].

Since the SPEC-OMP2001 benchmarks are highly parallel and represent only a small fraction of the application space, in this paper we also measure a synthetic benchmark written by the authors. The synthetic benchmark mimics applications with an adjustable ratio of parallel to serial code. It allows us to get accurate results within a short runtime, making it practical for exploring various scheduling options for various combinations of applications running together in the system. The synthetic benchmark consists of a loop of a mathematical calculation which fits entirely in the cache. During the course of its execution, the benchmark switches randomly between serial phases, in which there is only one active thread, and parallel phases, in which there are n threads, equal to the number of cores in the multiprocessor.

We model and label multithreaded programs by the ratio of parallel and serial instructions they contain, divided by the computing power of the multiprocessor on which they are run. In the following equation, I_P denotes the number of dynamic instructions executed in the parallel phases, I_S denotes the number of dynamic instructions executed in the serial phases, n denotes the number of cores in the multiprocessor, and the normalization factor k is chosen so that one of the ratios equals one, and the other is greater than or equal to one. The labeling is dependent on the number of cores, in order to hint the amount of time spent in the parallel and serial phases.

$$(ratio_{Parallel}, ratio_{Serial}) = \left(k\frac{I_{Parallel}}{n}, kI_{Serial} \right) \qquad (1)$$

For example, a benchmark labeled (1:1) on a symmetric CMP with no synchronization and scheduling overheads will spend roughly equal time in its parallel phases and in its serial phases. Completely parallel applications are labeled as (∞:1), whereas completely serial applications are labeled as (1:∞).

The synthetic benchmark may be tuned so that in the long run it would mimic the parallelism behavior of applications, ranging from completely parallel applications (∞:1) to completely serial applications (1:∞). Each measurement of the synthetic benchmark lasts 60 seconds, after which the benchmark reports the total number of iterations it has completed in that time frame. The pseudo-code of the synthetic benchmark is detailed in Fig. 3.

```
while (time<60 seconds) {
    parallel_iterations = random();
    serial_iterations = parallel_iterations * ratio_Serial /    (ratio_parallel * N_cores);
    in each thread { // fork
        if (calculated_iterations < parallel_iterations) {
            for (i=0;i<CHUNK;i++)
                perform_calculation();
            calculated_iterations += CHUNK; //shared variable
        }
    } //join
    for (i=0;i<serial_iterations;i++,calculated_iterations++)
        perform_calculation();
}
print "performance=",calculated_iterations/(time-start_time)
```

Fig. 3. Pseudo-code of the synthetic benchmark.

4 Methodology

This research is focused on the interactions between multiple multithreaded applications. All measurements in this paper, except where stated otherwise, are therefore performed for two applications running in parallel. We focus on three metrics: performance, fairness, and jitter.

Measuring the performance improvement of multiple applications running in parallel in different environments, for example, environments with the same hardware but with different OS schedulers, is not trivial [19]. The task is even harder when these applications are also multithreaded. Alameldeen et al. [2] have shown that the throughput metric of IPC used in uniprocessors is not accurate for multithreaded programs in multicore architectures. One of the reasons is that threads in a multithreaded program use polling when waiting for sibling threads, resulting in different number of committed instructions in different executions of the same program. The accurate throughput metric in multithreaded programs is therefore the amount of actual work performed divided by the execution time.

Measuring the throughput of two synthetic benchmarks running simultaneously is done by summing the number of iterations completed in each benchmark during a predefined benchmark time. The SPEC-OMP benchmarks, however, must run until completion, since they report their accurate progress only when done.

One way of measuring performance of a scheduler for multithreaded applications is to run two applications and wait for both to finish:

$$t_{completion} = \max(t_{Benchmark-1}, t_{Benchmark-2}) \quad (2)$$

This method is demonstrated in Fig. 4, and is similar to the "Last" method described in [19]. While measuring with this method, we found that in many cases one application finished its execution well before the other. Since we want to measure the interactions between applications, the time segment in which only one application is active becomes irrelevant, but it does affect the results.

Another option for measuring is to repeat short applications in order to equalize the runtimes, such as repeating application "A" in Fig. 4 twice. Equalizing runtimes, however, requires large numbers of iterations of long benchmarks. Additionally, while runtimes may be equal in one environment, such as the same system with a modified thread scheduler, they may be completely different in another.

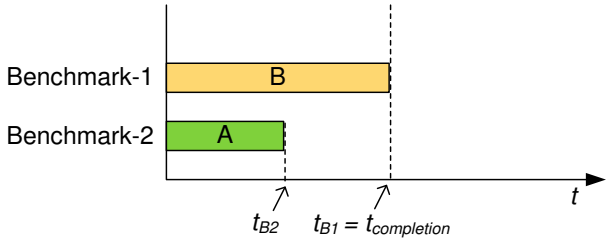

Fig. 4. Example of two multithreaded benchmarks running in parallel [19].

We handle the throughput measurement problem by running two benchmarks that perform the same work, each comprised of two applications that are run in a different order, as shown in Fig. 5. Since the work of the two benchmarks is the same, the runtimes are closer than in the previous methods. As a result, the effects of our new scheduling mechanisms can be evaluated more reliably than in the other methods [19].

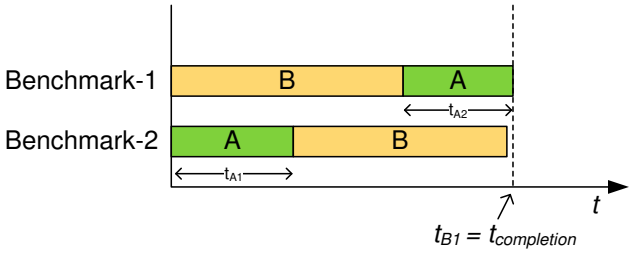

Fig. 5. Example of two multithreaded benchmarks running in parallel, each comprising two applications in a different order, allowing for closer benchmark execution times t_{B1} and t_{B2}.

The second metric evaluated in this paper is fairness. When two applications are executed in parallel, their runtimes are longer than when each application runs alone on the multiprocessor:

$$speedup_A = \frac{Performance_{A,A\|B}}{Performance_{A,A}} \quad (3)$$

The speedup values are actually less than one, representing a slowdown of application A because of sharing the system resources with application B. If both applications exhibit the same relative speedup, the system is said to be fair [9][17]. If, however, each application exhibits a different relative speedup, the system is unfair. In this paper, we use the fairness metric detailed in [9], which is defined as the minimum ratio of speedups of the applications. For two applications, fairness is defined as follows:

$$F = \min\left(\frac{speedup_A}{speedup_B}, \frac{speedup_B}{speedup_A}\right) \quad (4)$$

Fairness as defined above can be in the range of 0 to 1, corresponding to completely unfair and to completely fair, respectively. As shown in Fig. 5, we measured the SPEC-OMP2001 benchmarks by executing both applications twice but in a different order. The speedup of application A in equation (3) is therefore the time required to execute the application alone on the multiprocessor, divided by the average duration of application A in the configuration shown in Fig. 5:

$$speedup_A = \frac{t_{A,alone}}{0.5\left(t_{A1} + t_{A2}\right)} \quad (5)$$

The third metric is jitter in execution runtimes. In the ideal case, consecutive executions of the same applications in the same environment are expected to yield similar execution times. Balakrishnan et al. [5] have already shown, however, that operating system schedulers in asymmetric multiprocessors present unpredictable application runtimes for a single multithreaded application. In this paper, we quantify runtime jitter by measuring the standard deviation of the normalized execution times of the workload in N experiments of the same benchmark:

$$Jitter_A = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(\frac{t_{A,n}}{t_{A,avg}} - 1\right)^2} \quad (6)$$

5 Analysis

In this section, we analyze the performance, fairness and jitter metrics for two multithreaded applications, App_A and App_B, running in parallel. These applications have one active thread in their serial phases and n active threads in their parallel phases, which is also equal to the number of cores in the multiprocessor. The applications may differ in their parallel/serial ratios. We validate the analysis by measurements on a real multiprocessor with a modern OS.

Consider the example of an asymmetric multiprocessor with n cores: one is large with higher performance, whereas all others are small and with lower performance, as in [13]. The large core's performance is higher than the small cores' performance by the factor a. For simplicity, we assume that the performance factor a is identical for all workloads. In this section, the performance figures are normalized to the performance of one thread on one small core.

We assume for this analysis that there is equal probability for threads to run on any core of the multiprocessor. The average performance of a serial thread on an idle asymmetric multiprocessor is therefore given by:

$$Perf_{serial} \equiv P_S = \frac{1}{n}\cdot a + \frac{n-1}{n}\cdot 1 = \frac{n-1+a}{n} \quad (7)$$

The performance of a parallel application when running on an idle asymmetric multiprocessor is as follows:

$$Perf_{parallel} \equiv P_P = n - 1 + a = n \cdot P_S \quad (8)$$

We consider the case where two concurrently running applications have n active threads in their parallel phases. When both applications are serial, the scheduler has three options for scheduling. In the first option, both applications are scheduled on small cores. In the second and third options, one application is scheduled to run on the large core, and the other application is scheduled to run on a small core. There is also a fourth option in which both applications can run on the large core. This would make sense for large values of a ($a \geq 2$), which would allow better performance than would be achieved by scheduling each of the threads alone on their own small core. For simplicity, we assume that a is sufficiently small ($1 \leq a < 2$) in our analysis. The maximum performance in this case will be achieved when the serial application runs on the large core:

$$Perf_{(S,S),\max} = a \quad (9)$$

The minimum speedup will be achieved when the serial application runs on the small core:

$$Perf_{(S,S),\min} = 1 \quad (10)$$

Given the probabilities for each of the discussed cases, the applications will exhibit the following average performance:

$$Perf_{(S,S),avg} = \left(\frac{n-1}{n}\frac{1}{n-1}\right)a + \left(1 - \frac{n-1}{n}\frac{1}{n-1}\right)\cdot 1 = \frac{n-1+a}{n} = P_S \quad (11)$$

Thus, in the state (S,S) in the baseline scheduler, each application is indifferent to the existence of the other in the average case.

When both applications are in their parallel phases, there are exactly $2n$ running threads that compete for n cores. In this case there are three scheduling

options. In the first option, the scheduler schedules one thread of each application on the large core, and all other cores run two threads each. In the second and third options, only one application is scheduled to the large core.

When two threads share the same core, we assume that each has access to half of the computing resources of that core. The maximum performance for a parallel application will therefore be achieved when two threads of the parallel application are scheduled on the large core:

$$Perf_{(P,P),\max} = \frac{n-1+a}{2} - \frac{1}{2} + \frac{a}{2} = \frac{n-2+2a}{2} \quad (12)$$

The minimum speedup will be achieved when all threads of the parallel application are scheduled on the small cores:

$$Perf_{(P,P),\min} = \frac{n-1+a}{2} - \frac{a}{2} + \frac{1}{2} = \frac{n}{2} \quad (13)$$

The parallel applications will therefore exhibit the following average speedup:

$$Perf_{(P,P),avg} = \left(\frac{n}{2n}\frac{n-1}{2n-1}\right)\frac{n-2+2a}{2} + \left(2\frac{n}{2n}\frac{n}{2n-1}\right)\frac{n-1+a}{2} + \left(\frac{n}{2n}\frac{n-1}{2n-1}\right)\frac{n}{2} = \frac{n-1+a}{2} = \frac{1}{2}P_P \quad (14)$$

When one of the applications is serial and the other is parallel, there are $n+1$ threads that are to be scheduled on n cores. Out of the $n+1$ threads, exactly two threads will share a single core, and $n-1$ threads will each have their own cores. On the asymmetric multiprocessor, the maximum performance will be achieved when the serial application runs alone on the large core:

$$Perf_{(S,P),\max} = a \quad (15)$$

The minimum performance will be achieved when the serial application is run on a small core along with one of the threads of the parallel application:

$$Perf_{(S,P),\min} = \frac{1}{2} \quad (16)$$

For simplicity, we assume that all threads have equal probability to execute on the two-thread core and on the one-thread core. The probability of the serial thread sharing its core with another thread is therefore $2(n-1)^{-1}$, and the probability that this core is the large core is n^{-1}. The average performance of the serial thread when running concurrently with a parallel application is thus:

$$Perf_{(S,P),avg} = \frac{2}{n+1}\left(\frac{1}{n}\cdot\frac{a}{2} + \frac{n-1}{n}\cdot\frac{1}{2}\right) + \frac{n-1}{n+1}\left(\frac{1}{n}\cdot a + \frac{n-1}{n}\cdot 1\right) = \frac{n-1+a}{n+1} = \frac{n}{n+1}P_S \quad (17)$$

When a parallel application is running on an asymmetric multiprocessor concurrently with a serial thread, the maximum performance is achieved when the serial thread is scheduled together with one of the parallel threads on one of the small cores:

$$Perf_{(P,S),\max} = n - 3/2 + a \quad (18)$$

The minimum performance is achieved when the serial thread is scheduled alone on the large core:

$$Perf_{(P,S),\min} = n-1 \quad (19)$$

The average performance of the parallel application when running simultaneously with a serial application is given by:

$$Perf_{(P,S),avg} = \frac{2}{n+1}\left(\frac{1}{n}\cdot(n-1+a/2) + \frac{n-1}{n}\cdot(n-3/2+a)\right) + \frac{n-1}{n+1}\left(\frac{1}{n}\cdot(n-1) + \frac{n-1}{n}\cdot(n-2+a)\right) = \frac{n}{n+1}P_P \quad (20)$$

The average speedup, calculated according to (3), and the maximum and minimum values in each state are shown in Table 1.

Table 1 – Baseline scheduler: Speedups (Min, Average, Max) for application "A" on the asymmetric multiprocessor. *n*=Number of cores. *a*=Performance of the large core.

Case (A,B)	Minimum Speedup	Average Speedup	Maximum Speedup	Maximum / Minimum
S,S	$\frac{n}{n-1+a}$	1	$a\frac{n}{n-1+a}$	a
S,P	$\frac{1}{2}\frac{n}{n-1+a}$	$\frac{n}{(n+1)}$	$a\frac{n}{n-1+a}$	$2a$
P,S	$\frac{n-1}{n-1+a}$	$\frac{n}{(n+1)}$	$\frac{n-3/2+a}{n-1+a}$	$\frac{n-3/2+a}{n-1}$
P,P	$\frac{n}{2(n-1+a)}$	$\frac{1}{2}$	$\frac{n-2+2a}{2(n-1+a)}$	$\frac{n-2+2a}{n}$

The number of phase shifts between parallel and serial phases in an application, their timing, as well as

the length of each phase, may differ in different applications. When two applications with long phases are executed in parallel, the initial scheduling made by operating system schedulers at the beginning of the long phases therefore has a great impact on performance. For example, an initial scheduling might place the serial thread of an application together with one of the parallel threads of another application. If the phases are long, the serial application will exhibit a significant slowdown for a long period of time.

In order to verify the predicted speedups in Table 1, we measured a fully parallel synthetic benchmark running simultaneously with a fully serial synthetic benchmark. Using the affinity property in Linux, we were able to emulate multiprocessors with less than the eight physical cores our multiprocessor contains, by confining our benchmarks to a predefined set of cores. Since the results are sensitive to the initial state of the scheduler, the different threads of the application were first scheduled to randomly-chosen cores, and were then migrated to other cores by the Linux kernel load balancing mechanism.

The results for 50 runs for each symmetric multiprocessor configuration ($a=1$) are shown in Fig. 6. The horizontal axis is the number of cores and the number of threads in the parallel application, whereas the vertical axis is the speedup of the serial application. The average measured speedups are close to the theoretical predictions and converge to 1 as n increases. The differences between the theoretical and measured values are caused by the different initial scheduling made by the baseline scheduler in comparison with the assumed initial scheduling in the theoretical equations. The error bars in Fig. 6 show that the range of possible speedups is between 0.5 and 1, in accordance with our expectations. The other three states in Table 1: (S,S), (P,S), and (P,P), were measured in the same manner and were also in line with the analytic predictions.

Fairness is calculated according to (4), and is summarized in Table 2. When two applications are in their serial phase, in the worst case the fairness is given by dividing the minimum and maximum speedups of the state (S,S).

When one application is serial and the other is parallel, there are two cases for fairness. In the first case, the serial application exhibits the minimum performance shown in (16), and the parallel application exhibits its maximum performance shown in (18). The fairness in this state in the worst case is therefore given by dividing the minimum speedup in the state (S,P) by the maximum speedup in state (P,S).

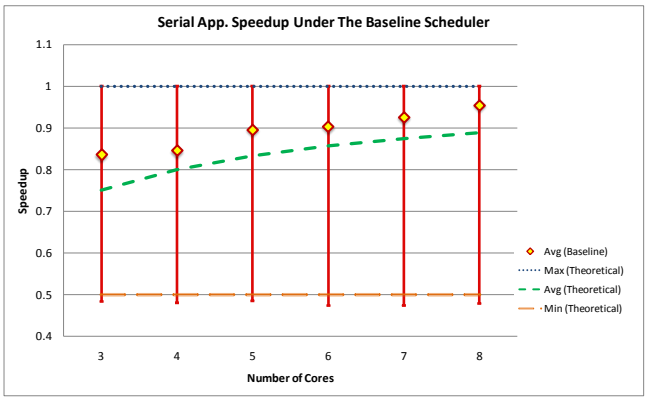

Fig. 6. Symmetric multiprocessor ($a=1$): The measured speedup of the serial synthetic benchmark (1:∞) when run in conjunction with the parallel synthetic benchmark (∞:1).

When the serial application exhibits the maximum performance shown in (15), the parallel application exhibits its minimum performance shown in (19). The fairness of this state in the worst case is therefore given by dividing the minimum performance in state (P,S) by the maximum performance in state (S,P).

When both applications are in their parallel phases, the fairness in the worst case is given by dividing the minimum speedup by the maximum speedup in the state (P,P).

Table 2 – Worst case fairness equations for the baseline scheduler.

(S,S)	(S,P),(P,S) case 1	(S,P),(P,S) case 2	(P,P)
$\frac{1}{a}$	$\frac{n}{2(n-3/2+a)}$	$\frac{n-1}{an}$	$\frac{n}{n-2+2a}$

The results from Table 1 and the worst case fairness equations in Table 2 indicate that as the ratio between the performance of the cores in the asymmetric multiprocessor increases (a), the fairness in the worst case decreases and the jitter between runtimes increases.

The analysis in this section reveals that in current operating system schedulers, which are not application phase aware, applications may exhibit different speedups owing to the interactions between the applications in their different phases. Applications in their serial phase may be slowed down by a factor of up to two on symmetric multiprocessors when running simultaneously with applications in their parallel phase. As a result, these schedulers are unfair

and may produce jitter in execution runtimes of applications.

6 Proposed Scheduling Algorithm

We propose a new thread scheduling algorithm that aims to improve performance, improve fairness and reduce the jitter in execution runtimes. The proposed algorithm grants higher scheduling priority to the serial threads. As a result, when a serial thread is executed concurrently with a parallel application, the serial thread is always granted a core for itself. The scheduling mechanism results in the minimum and maximum speedups shown in Table 3.

Table 3 – Minimum and maximum speedups of application A for the proposed scheduler on asymmetric multiprocessors.

Case (A,B)	Minimum Speedup	Average Speedup	Maximum Speedup	Maximum / Minimum
S,S	$\frac{n}{n-1+a}$	1	$a\frac{n}{n-1+a}$	a
S,P	$a\frac{n}{n-1+a}$	$a\frac{n}{n-1+a}$	$a\frac{n}{n-1+a}$	1
P,S	$\frac{n-1}{n-1+a}$	$\frac{n-1}{n-1+a}$	$\frac{n-1}{n-1+a}$	1
P,P	$\frac{n}{2(n-1+a)}$	$\frac{1}{2}$	$\frac{n-2+2a}{2(n-1+a)}$	$\frac{n-2+2a}{n}$

In state (S,S) on the asymmetric multiprocessor, there are two active serial threads but only one of them is granted the large core. This presents jitter in execution times, which could be avoided for example by the method proposed by Fedorova et al. [8] at the expense of thread migrations. Another possible method is to grant priority for computing power per application and not per thread. Exploration of this issue is left for future work. State (P,P) is similar, and the jitter in this state could also be avoided by the same methods. In the states (S,P) and (P,S), the minimum and maximum speedups are identical.

For the symmetric case (*a=1*), our analysis predicts identical minimum and maximum execution times for each state, so that jitter will be minimized and fairness between applications will improve.

The Linux scheduler has been extended to detect whether an application is in its parallel phase or in its serial phase. This is done by keeping track of the number of ready threads in each thread group, and is performed in O(1) time whenever a thread changes its ready state. A thread group is considered parallel when it has more than two ready threads. We chose two as the threshold since we noticed that an Open-MP application would frequently switch between one and two active threads.

The scheduler was also extended to grant higher priority to serial threads. In Linux, each thread has a property named dynamic priority. When the dynamic priority figure of a thread is lower, the thread is granted more CPU time. The priority was therefore boosted by subtracting ten [1] from the dynamic priority property of the thread.

When at least two applications are in their parallel phases, and each has a number of active threads that is at least equal to or larger than the number of cores in the system, the applications compete with each other without any throughput gains. This competition results in many unnecessary context switches that thrash the cache and lower the overall throughput of the system. In order to avoid this situation, our proposed scheduler boosts the priority of the application that was the first to enter its parallel phase. We call this mechanism “seniority boost”, as the scheduler chooses the senior application and boosts its priority. As a result, the application with the seniority boost is expected to finish its parallel phase sooner, while the system exhibits fewer context switches. When one of the applications finishes its parallel phase, the system transitions to one of the joint states (P,S) or (S,P) and the seniority boost is terminated. In order to avoid starvation, following a specific timeout in state (P,P) the seniority boost is removed and applied to the other application.

Apart from the above, the baseline Linux scheduler's thread migration policy has been revised. Threads whose applications become serial are automatically rescheduled on the idlest core and granted more priority. In asymmetric configurations, the high priority given to these threads will usually result in migration to the high performance core.

The asymmetric multiprocessor is emulated by changing the frequency (duty cycle) of seven out of eight cores in our symmetric multiprocessor, as done in [5] and [10]. In our case, we chose $a = 2$, so the frequency of seven of the eight cores was halved. Additionally, we configured the scheduler to treat the large core as having more performance by using the Linux CPU group property “cpu_power”. As a result, the scheduler attempts to schedule more work on the large core than on the small cores.

In order to verify the performance equations for the proposed scheduler, we measured a fully parallel

synthetic benchmark running in parallel with a fully serial synthetic benchmark. The results for 50 runs for each symmetric multiprocessor configuration are shown in Fig. 7. The speedup converged to one as predicted in Table 3, and the jitter was eliminated, in comparison with Fig. 6. The other three states were measured as well in the same manner and were also in line with the analytic predictions. The results for asymmetric multiprocessors were also in line with our expectations, with the average, minimum and maximum performance figures all converging to *a*.

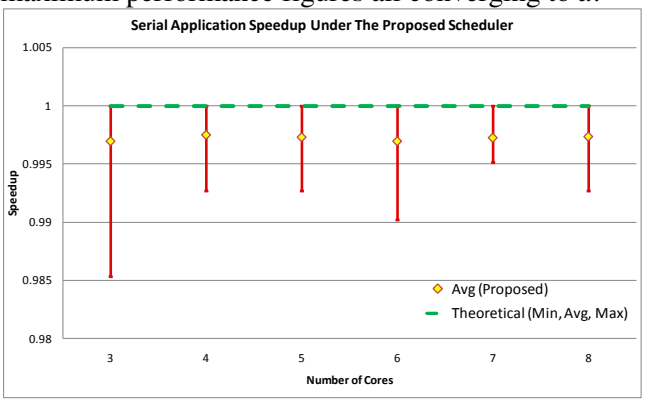

Fig. 7. The measured speedup of a serial application when running in conjunction with a parallel application under the proposed scheduler on symmetric multiprocessors.

7 Experimental Results

The idle time percentage measured in the synthetic benchmarks decreased as expected, from 20% to 17.2% (reduction by 14%) in the symmetric configuration, and from 25.6% to 22.8% (reduction by 10.9%). As a result, throughput improved by 3% and 4.5% respectively for the symmetric and asymmetric configurations, as shown in Table 4 for the asymmetric multiprocessor.

Table 4 – Asymmetric multiprocessors (*a=2*): The speedup of two concurrently running synthetic benchmarks when the proposed scheduler is used in comparison with the baseline Linux scheduler.

	(∞:1)								
(∞:1)	1%	(8:1)							
(8:1)	-1%	1%	(4:1)						
(4:1)	-1%	1%	1%	(2:1)					
(2:1)	0%	-1%	4%	4%	(1:1)				
(1:1)	-2%	1%	3%	4%	7%	(1:2)			
(1:2)	0%	1%	3%	5%	8%	7%	(1:4)		
(1:4)	-2%	2%	0%	6%	9%	11%	8%	(1:8)	
(1:8)	-2%	1%	3%	8%	7%	15%	18%	3%	(1:∞)
(1:∞)	-2%	2%	3%	6%	12%	16%	17%	10%	12%
AVG	-1%	1%	2%	4%	5%	7%	8%	7%	8%
Average speedup of all dual benchmarks: +4.5%									

Fig. 8 shows a contour graph of the speedup in the symmetric multiprocessor, with peak speedup at benchmarks (1:1) in parallel to (1:1). Speedups decrease monotonically when moving away from this peak.

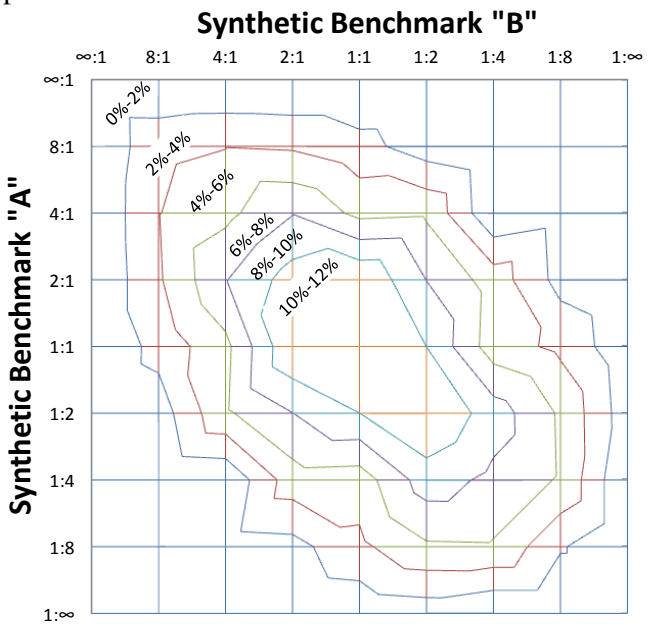

Fig. 8. Symmetric multiprocessor (*a=1*): The speedup of two concurrently running synthetic benchmarks when the proposed scheduler is used in comparison with the baseline Linux scheduler.

The average fairness and jitter metrics improved as well, as detailed in Table 5. The high fairness in the asymmetric configuration in the baseline scheduler was achieved due to the awareness of the baseline scheduler for the asymmetry, as explained in section 5. This awareness allowed the scheduler to schedule more work on the large core, thereby granting similar computing power to all threads.

The jitter shown in Table 5 is multiplied by 1000, and was reduced on average by 60% in the symmetric case and by 35% in the asymmetric case.

Table 5 – The average fairness and jitter metrics with the baseline and proposed schedulers, for the synthetic benchmarks.

	Symmetric		Asymmetric	
Scheduler	Fairness	Jitter	Fairness	Jitter
Baseline	75.9%	9.07	87.5%	38.74
Proposed	90.7%	3.66	88.7%	25.12
Improvement	**19.5%**	**59.7%**	**1.4%**	**35.1%**

Table 6 shows the speedup for the SPEC-OMP2001 benchmarks with the proposed scheduler, in comparison with the baseline Linux scheduler. The measurements were performed according to the

method shown in Fig. 5. The speedup exhibited by the highly parallel SPEC-OpenMP benchmarks averaged 1.5% in the symmetric multiprocessor, and 3.5% in the asymmetric multiprocessor, as shown in Table 6 for the asymmetric multiprocessor.

Table 6 – The speedup of two concurrently running SPEC-OMP2001 benchmarks with the proposed scheduler in comparison with the baseline Linux scheduler in the asymmetric configuration.

	wup									
wupwise	2%	swi								
swim	8%	1%	mgr							
mgrid	4%	4%	4%	app						
applu	2%	-2%	3%	1%	equ					
equake	3%	0%	4%	0%	0%	aps				
apsi	12%	15%	7%	12%	9%	16%	gaf			
gafort	1%	3%	2%	-2%	2%	7%	0%	fma		
fma3d	1%	5%	3%	-3%	0%	9%	3%	3%	art	
art	2%	0%	4%	3%	-1%	15%	-1%	2%	-1%	amm
ammp	3%	1%	4%	-2%	4%	13%	0%	4%	1%	1%
average	**4%**	**3%**	**4%**	**1%**	**2%**	**11%**	**1%**	**3%**	**2%**	**3%**
Average speedup of all dual benchmarks: 3.5%										

The average fairness and jitter metrics for the SPEC-OMP benchmarks are shown in Table 7. The jitter, measured on 5 runs of "equake" & "art" as an example, was almost eliminated in the symmetric case and was halved in the asymmetric case.

Table 7 – The average fairness and jitter metrics with the baseline and proposed schedulers, for the SPEC-OMP benchmarks.

	Symmetric		Asymmetric	
Scheduler	Fairness	Jitter	Fairness	Jitter
Baseline	79.6%	1.13	49.3%	1.90
Proposed	78.5%	0.13	62.1%	0.94
Improvement	**-1.4%**	**88.1%**	**25.9%**	**50.5%**

The SPEC-OMP2001 benchmarks are highly parallel applications, corresponding roughly to the (∞:1) and (8:1) synthetic benchmarks. Their speedups are therefore similar, in the ranges 1-3% for the symmetric case and 3-4% for the asymmetric case. According to the synthetic benchmark results, we predict that applications with lower parallelism will exhibit higher throughput using our proposed scheduler.

8 Conclusions and Future Work

In this paper, we proposed a new scheduling mechanism that favors serial phases of applications over parallel phases. When running two multithreaded scientific applications (SPEC-OMP2001) on symmetric as well as on asymmetric multiprocessors, analytical and experimental results showed improvements in all metrics; the jitter in execution runtimes decreased by as much as 88%, throughput in some cases increased by more than 16%, and the fairness metric improved by up to 26%.

The experiments in this paper were performed on a real system, using official benchmarks and a modern operating system (Linux kernel 2.6.18) with our extensions. The exhibited performance improvements therefore are system-wide, taking into account all factors such as cache contention, memory subsystem, as well as a complete software stack and operating system. The concepts presented in this paper could easily be implemented in today's state of the art multiprocessor operating systems, as implemented in our experimental system, and could show immediate performance gains.

This work provides insights into a multitude of future research issues in the area of multithreaded application handling in CMP. While our results indeed show improvements, some benchmarks were significantly faster whereas others exhibited a slowdown with our proposed scheduler. In future work, this phenomenon should be explored more deeply, possibly resulting in an adaptive mechanism that could improve throughput even further.

The analysis in this paper could be extended to take into account the distribution of phase-changing during the runtime of applications. Additionally, the way multithreaded programs were modeled in this paper, having either one active thread or *n* active threads, could be extended to include the whole range from one to *n*. Such extensions could later be used to improve system metrics even further, even on current symmetric architectures.

With regards to asymmetric configurations, the analysis in this paper could be extended to support various configurations of asymmetric multiprocessors, such as more than two types of cores. Additionally, the analysis could take into account different speedups for different applications on each core type.

Acknowledgements

We thank Avi Mendelson and Andrey Gelman for their insightful comments. We also thank Andrey Gelman and Niv Aibester for their help in setting up the emulation environment.

References

[1] J. Aas. "Understanding the Linux 2.6.8.1 CPU scheduler." SGI, 2005, http://josh.trancesoftware.com/linux/

[2] A.R. Alameldeen and D.A. Wood. “IPC Considered Harmful for Multiprocessor Workloads.” In IEEE Micro, Jul-Aug 2006.

[3] M. Annavaram, E. Grochowski, and J. Shen. “Mitigating Amdahl’s Law Through EPI Throttling.” In Proc. of the 35th ISCA, June 2005.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones, and B. Parady. “SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance.” In Proc. of WOMPAT 2001.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. “The Impact of Performance Asymmetry in Emerging Multicore Architectures.” In Proc. of the 35th ISCA, June 2005.

[6] F.A. Bower, D.J. Sorin, and L.P. Cox. “The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling.” IEEE Micro, May/June 2008.

[7] S. Chen, P.B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry, and C. Wilkerson. “Scheduling Threads for Constructive Sharing on CMPs.” In Proc. of ACM SPAA 2007.

[8] A. Fedorova, D. Vengerov, and D. Doucette. “Operating System Scheduling On Heterogeneous Core Systems.” In Proc. of the Operating System support for Heterogeneous Multicore Architectures (OSHMA) workshop, 16th PACT, September 2007.

[9] R. Gabor, S. Weiss, and A. Mendelson. “Fairness and Throughput in Switch on Event Multithreading.” In Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006.

[10] E. Grochowski, R. Ronen, J. Shen, and H. Wang. “Best of Both Latency and Throughput.” In Proc. of the 22nd ICCD, October 2004.

[11] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. “Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance.” In Proc. of the 31st ISCA, June 2004.

[12] K. Luo, J. Gummaraju, and M. Franklin. “Balancing throughput and fairness in SMT processors.” In Proc. of the ISPASS, pages 164–171, 2001.

[13] T.Y. Morad, U.C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé. “Performance, Power Efficiency, and Scalability of Asymmetric Cluster Chip Multiprocessors.” In Computer Architecture Letters, vol. 4, 2005.

[14] O. Mutlu and T. Moscibroda. “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors.” In Proc. of the 27th PODC, 2008.

[15] K. Olukotun and L. Hammond. “The future of microprocessors.” In ACM Queue, vol. 3, no. 7, 2005.

[16] OpenMP Architecture Review Board. “OpenMP Application Program Interface.” http://www.openmp.org, version 2.5, May 2005.

[17] S.E. Raasch and S. K. Reinhardt. “Applications of Thread Prioritization in SMT Processors.” In Proc. 1999 Workshop on Multithreaded Execution And Compilation, 1999.

[18] A. Snavely, D. Tullsen, and G. Voelker. “Symbiotic jobscheduling with priorities for a simultaneous multithreading processor.” In proc. of the 2002 ACM SIGMETRICS, 2002.

[19] J. Vera, F.J. Cazorla, A. Pajuelo, O.J. Santana, E. Fernández and M. Valero. “FAME: Fairly MEasuring Multithreaded Architectures”. IEEE-ACM PACT Conference, Parallel Architectures and Compilation Techniques. Brasov, Romania, September 15-19, 2007.

[20] J.A. Winter and D.H. Albonesi. “Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures.” In proc. of the 38th DSN, June 2008.