Scheduling 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, fairness and jitter of workloads consisting of multiple multithreaded applications running on Chip MultiProcessors (CMP). Current thread assignment techniques which are tailored for single-thread applications result in sub-optimal usage of the multiprocessor resources, unfairness between applications and jitter in execution runtimes when dealing with multiple multithreaded applications running in parallel. Multithreaded applications contain serial phases (single thread) and parallel phases (many threads). In this paper, we propose a new thread assignment mechanism that takes into account the different requirements of each phase, granting higher priority to applications during their critical-serial phases. Analytic and experimental evaluation of the proposed thread assignment mechanism on both symmetric and asymmetric multiprocessors show throughput improvements by as much as 16%, improved fairness by as much as 26% and reduced jitter by as much as 88%.**

Keywords- Asymmetric Multiprocessors, Operating Systems, Scheduling.

I. 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 [17], 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 several multithreaded applications are run in parallel on symmetric multiprocessors, where all cores are identical, as well as on asymmetric multiprocessors [12], where some computing cores are faster than others.

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.

When several multithreaded applications run simultaneously on a multiprocessor, the serial thread of one application may be available for execution together with the threads of the other applications. Fig. 1 shows an example of the four possible joint states of two multithreaded applications. The vertical axis represents time, and the number of active threads of each application is shown for each point in time.

Current thread assignment techniques, such as the technique used in 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 (unpredictable performance), 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.

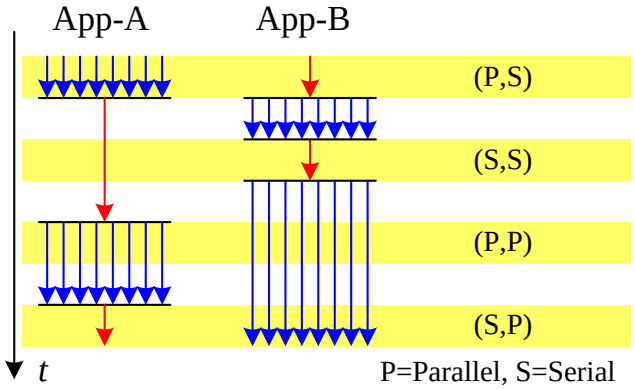

Fig. 1. Illustration of joint states of two sample applications running simultaneously.

In this paper, we propose to add another dimension to the current thread assignment mechanisms by using information about the parallel and serial phases of applications. Our proposed thread assignment technique 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 the state transition arcs denote the most

likely transition. The proposed thread assignment technique, shown in Fig. 2b, favors the serial thread, thus increasing the probability for transition from (S,P) and (P,S) states to (P,P) state. Current OS schedulers (shown in Fig. 2a), however, treat the serial and parallel threads equally, thereby lengthening the time required for the serial application to transition into its parallel phase. The proposed technique is expected to improve throughput by reducing the time spent in (S,S) in which there are idle cores, resulting in increased core utilization.

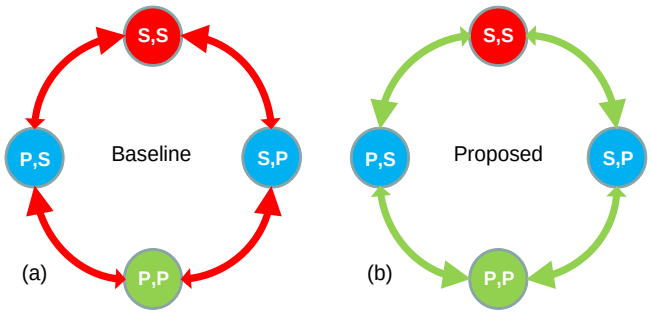

Fig. 2. Illustration of the four possible joint states of two applications running in parallel.

In this paper, we propose a new thread assignment technique 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, and with workloads consisting of 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.

II. Related Work

There are a number of papers addressing the scheduling of single-threaded applications on asymmetric/heterogeneous multiprocessors [12], which are based on sampling of runtime performance on the different core types. Kumar et al. [13] have proposed a scheduler for multiple single-threaded applications on a heterogeneous multiprocessor. Bower et al. [7] have shown the impact on thread scheduling in symmetric multiprocessors that become heterogeneous during runtime owing to frequency scaling, process variations and physical faults. Winter et al. [24] explored thread assignment algorithms for single-thread applications on such multiprocessors.

Other papers address the scheduling problem of a single multithreaded application running on an asymmetric multiprocessor [4][6][11][15]. Grochowski et al. [4][11] have proposed a static scheduling mechanism, implemented at the application level, which schedules the serial phases of applications on the high performance core. Balakrishnan et al. [6] 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 [4][6][11] for multiple multithreaded programs, while addressing the scheduling problem that arises when there are more threads than cores in the multiprocessor. Our results are compared versus a baseline environment without the proposals from previous work, since these proposals are tailored for single multithreaded applications and therefore will perform similarly to the baseline environment when more than one multithreaded application is run in parallel.

Many papers explore fairness and throughput in SMT architectures [10][14][19][21]. We use and extend their throughput and fairness metrics for asymmetric multiprocessors. Other papers explore scheduling multiple multithreaded applications on symmetric systems [3][16][23]. We extend these ideas for asymmetric configurations and present the ability to prioritize applications based on their phase of execution.

III. Emulation Environment

All measurements in this paper are performed on an 8-core multiprocessor (HP ProLiant DL580) consisting of four dual-core 2.66GHz Intel Xeon processors (7020), 667MHz front side bus, 8GB of DDR2 memory, and with SMT disabled for better emulation of symmetric systems. The operating system used is Linux 2.6.18, and is referred to in this paper as the baseline environment. Our benchmarks include the entire SPEC-OMP2001 [5] 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 [18]. 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 [6] and [15], and allows higher core utilization on heterogeneous multiprocessors.

Since the SPEC-OMP2001 benchmarks are highly parallel and represent only a small fraction of the application space, we also measure in this paper 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 that 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 number of cores used in each phase. In the following equation, I_P and I_S denote the number of dynamic instructions executed in the parallel and serial phases respectively, 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. For simplicity, we assume identical IPC for the parallel and serial phases.

$$(ratio_{Parallel}, ratio_{Serial}) = \left(k\frac{I_P}{n}, kI_S \right) \quad (1)$$

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.

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.

IV. Methodology

This research is focused on the interactions between multiple multithreaded applications that are run in parallel. In particular, 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 no trivial task [22]. It is even harder when the applications are multithreaded. Alameldeen and Wood [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 for this is that threads in a multithreaded program use polling when waiting for sibling threads, resulting in a different number of committed instructions in different executions of the same program. The accurate throughput metric for multithreaded programs is therefore the amount of actual work performed divided by the execution time.

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

One way of measuring the throughput of a thread assignment mechanism for multithreaded applications is to run two applications and wait for both to finish. This method is demonstrated in Fig. 3, and is similar to the "Last" method described in [22]. 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.

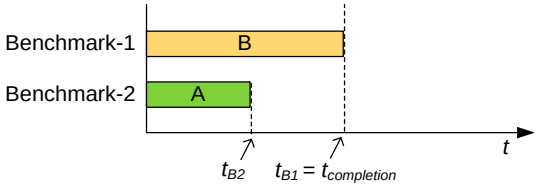

Fig. 3. Example of two multithreaded benchmarks running in parallel.

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. 4. Since the work of the two benchmarks is identical, the runtimes are closer than in the previous methods. As a result, the effects of our new scheduler can be evaluated more reliably than in the other methods [22].

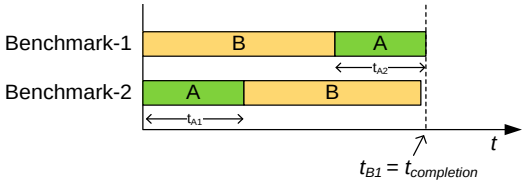

Fig. 4. Two multithreaded benchmarks, each comprising two applications in a different order, with closer execution times t_{B1} and t_{B2}.

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:

$$speedup_A = \frac{Performance_{A,A||B}}{Performance_{A,A}} \quad (2)$$

If both applications exhibit the same relative speedup, the system is said to be fair [10][19]. In this paper, we use the fairness metric detailed in [10], which is defined as the minimum ratio of speedups of the applications. For two applications, fairness is defined as follows:

$$Fairness_{A||B} = \min\left(\frac{speedup_A}{speedup_B}, \frac{speedup_B}{speedup_A} \right) \quad (3)$$

Fairness as defined above can be in the range of zero to one, corresponding to completely unfair and to completely fair, respectively. We calculate the speedup of application "A" in equation (2) as 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. 4: $t_{A,alone} / 0.5(t_{A1}+t_{A2})$.

The third metric we consider is jitter in execution runtimes. Balakrishnan et al. [6] have already shown 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:

$$Jitter_A = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(\frac{t_{A,n}}{t_{A,avg}} - 1\right)^2} \quad (4)$$

V. Analysis

In this section, we analyze the performance, fairness, and jitter metrics for multiple multithreaded applications 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. This assumption does not always hold in reality, since not all threads reach their barriers at the same time. When using dynamic work distribution with workloads consisting of long parallel phases, however, the effects of having less than *n* active threads in the parallel phases can be neglected. The applications in this model may differ in their parallel/serial ratios. The performance figures in this section are normalized to the performance of one thread on one small core.

We begin by considering two multithreaded applications, App_A and App_B, running on a symmetric multiprocessor with *n* identical cores. When both applications are in their serial phase, they are indifferent to each other's existence, since the scheduler would schedule each thread on a different core. Thus, two serial threads will exhibit no slowdown, and *n-2* cores will be idle. When both applications are in their parallel phases, there are *2n* running threads that compete for *n* cores. The Linux scheduler will schedule two threads on each core, thereby slowing down each application by a factor of two, assuming that the threads have equal priority and are not IO bound.

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, two threads will share a single core, and *n-1* threads will each have their own core. In the worst case for the serial application, it will be assigned to run with another thread, thus receiving only half of the computing power. In the best case for the serial application, it will exhibit no slowdown as it will be scheduled to run on a core by itself.

For simplicity, we assume that all threads have equal probability to execute on the two-thread core and on the one-thread core. The average speedup is therefore:

$$Speedup_{serial} = \frac{2}{n+1}\cdot\frac{1}{2} + \left(1-\frac{2}{n+1}\right)\cdot 1 = \frac{n}{n+1} \quad (5)$$

Since the probabilities of each thread to execute on each core are identical, the average speedup of the parallel phase is identical to that of the serial phase.

We extend the analysis for an asymmetric multiprocessor with *n* cores: one of the cores is larger as in [15], and is faster by a factor (*a*). For simplicity, we assume that the performance factor (*a*) is identical for all workloads. The average speedup, calculated according to (2), and the maximum and minimum values in each state for the asymmetric multiprocessor are shown in Table 1.

Table 1. Speedups (Min, Average, Max) for application "A" in the baseline environment on the asymmetric multiprocessor. *n*=Number of cores. *a*=Performance ratio of the large core.

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

Fairness is calculated according to (3), and is summarized in Table 2. When two applications are both in their serial phase or both in their parallel phase, the lower bound for fairness is given by dividing the minimum and maximum speedups of the states (S,S) or (P,P) respectively. When one application is serial and the other is parallel, there are two cases for fairness. In the first case, the lower bound for fairness is given by dividing the minimum speedup in the state (S,P) by the maximum speedup in state (P,S). In the second case, the lower bound for fairness is given by dividing the minimum speedup in state (P,S) by the maximum speedup in state (S,P).

Table 2. Lower bound for fairness in the baseline environment.

(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 (*a*) increases, the lower bound for fairness decreases and the jitter between runtimes increases.

We extend the analysis in this section for $k>2$ multithreaded applications running in parallel. The extension results are detailed in Table 3. The "Serial" or "Parallel" rows in the table show the speedups for a serial or parallel

phase of an application under all possible phases of the other applications running in parallel, in comparison with it running alone. The analysis predicts that as the number of applications (k) that are run in parallel increases, the possible jitter widens.

The probability of having idle cores decreases exponentially as more parallel applications are run in parallel (S,S,S,…). Consequently, the throughput gains of using our mechanism are expected to decrease as the number of parallel applications that are run in parallel increases.

Table 3. Minimum and maximum speedups for $2<=k<=n$ applications running in parallel.

Phase	Minimum Speedup	Maximum Speedup	Maximum / Minimum
Serial	$\frac{1}{ka}$	1	ka
Parallel	$\frac{n}{k(n-1+a)}$	$\frac{n-1+a-\frac{k-1}{2}}{n-1+a}$	$k\frac{n-1+a-\frac{k-1}{2}}{n}$

VI. PROPOSED ALGORITHM

We propose a new thread assignment algorithm that aims to improve performance, improve fairness and reduce the jitter in execution runtimes. The proposed algorithm grants higher scheduling priority to serial threads. As a result, when a serial thread is executed concurrently with a parallel application, the serial thread is granted a core for itself, and the threads of the parallel application will compete for the remaining cores. The scheduling mechanism results in the speedups shown in Table 4.

Table 4. Minimum and maximum speedups of application "A" for the proposed thread assignment technique on asymmetric multiprocessors.

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

For the symmetric case (a=1), our analysis predicts identical minimum and maximum execution times for all states, so that jitter will be minimized and fairness between applications will improve using our proposed scheduler.

It is expected that the proposed scheduler will reduce the time in which there are idle cores on the multiprocessor. As a result, the cores will spend more time performing useful work, increasing overall multiprocessor throughput.

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 many thread migrations. Another possible method is to grant priority for computing power per application and not per thread. State (P,P) is similar, and the jitter in this state could also be avoided by using similar methods.

The predicted speedups of the proposed scheduler for more than two applications running in parallel are given in Table 5.

The Linux scheduler has been extended to support the proposed algorithm. Detection of whether an application is in its parallel phase or in its serial phase is done by keeping track of the number of ready threads in each thread group. This 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, and is considered serial otherwise. We chose two as the threshold since we noticed that an Open-MP application frequently switched between one and two active threads.

Table 5. Speedups for $2<=k<=n$ applications running in parallel using the proposed scheduler.

Phase	Minimum Speedup	Maximum Speedup	Maximum / Minimum
Serial	$\frac{1}{a}$	1	a
Parallel	$\frac{n}{k(n-1+a)}$	$\frac{n-k+1}{n-1+a}$	$k\frac{n-k+1}{n}$

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

The load balancer of the Linux kernel was extended as well; the baseline scheduler will not migrate a running thread, and will not migrate a ready thread from a slow core to a fast core if it is the only running thread on the slow core. These were changed to allow for better load balancing on asymmetric multiprocessors.

When at least two applications are in their parallel phases, and each has a number of active threads that is at least equal to the number of cores in the system, the applications compete with each other without any throughput gains. This competition, which is favored by our proposed technique, 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 technique 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. This mechanism is similar to gang scheduling [9][20]. When using this mechanism, 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 removed. In order to avoid starvation, following a specific timeout in state (P,P)

the seniority boost is removed and applied to the other application.

The baseline Linux scheduler's thread migration policy has also 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 [6] and [11]. 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 attempted to schedule more work on the larger core.

The proposed scheduler will require additional changes to perform well when there are more applications than cores, since serial threads may dominate the computing resources. Such changes may include a timeout for the bonus granted to serial threads.

VII. Experimental Results

The idle time percentage measured for two synthetic benchmarks running in parallel 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%) in the asymmetric configuration. This is in-line with our expectations that the multiprocessor's utilization will be increased with the proposed scheduler. Throughput improved by 3% and 4.5% respectively for the symmetric and asymmetric configurations, as shown in Table 6 for the asymmetric configuration.

Table 6. Measured speedup of two concurrently running synthetic benchmarks using the proposed technique on an asymmetric multiprocessor configuration (a=2).

	(∞: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. 5 shows a contour graph of the speedup in the symmetric multiprocessor. Each axis represents an application, ranging from completely serial (1:∞) to completely parallel (∞:1). The data in the graph corresponds to the speedup of the two applications running in parallel on a symmetric multiprocessor with the proposed scheduler, in comparison to the baseline scheduler. Peak speedup is achieved by the combination of benchmark (1:1) with a similar benchmark (1:1). Speedups decrease monotonically when moving away from this peak. The expected speedups of the highly parallel SPEC-OMP2001 benchmarks should roughly correspond to the (∞:1) and (8:1) benchmarks, which are between 0%-4% in the symmetric configuration.

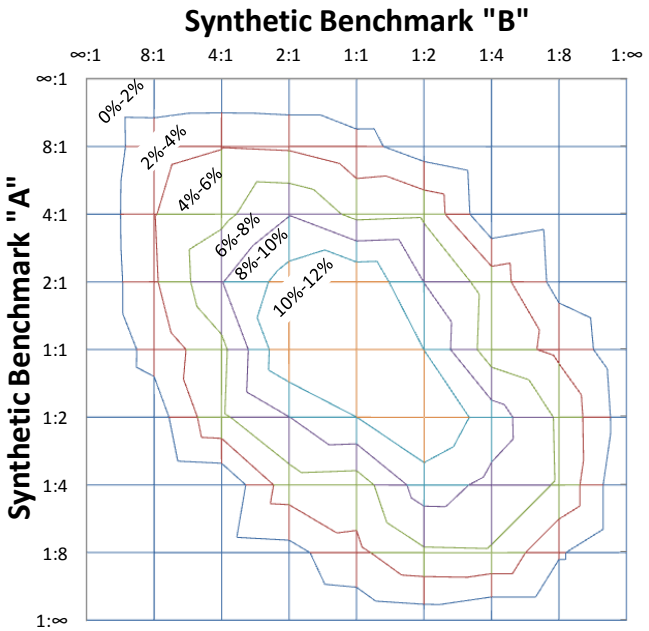

Fig. 5. Experimental contours of the speedup of two concurrently running synthetic benchmarks when the proposed technique is used on a symmetric multiprocessor configuration (a=1).

Table 7 shows the speedups for the SPEC-OMP2001 benchmarks with the proposed thread assignment technique. The measurements were performed according to the method shown in Fig. 4. The speedup exhibited by the highly parallel SPEC-OMP benchmarks averaged 1.5% in the symmetric multiprocessor, and 3.5% in the asymmetric multiprocessor. The "apsi" benchmark showed significant improvement because it had many phase shifts between parallel/serial phases. Since our proposed mechanism reacts fast to these frequent phase shifts, the bottlenecks of "apsi" were freed faster, and hence "apsi" achieved greater speedups.

Table 7. Measured speedup of two concurrently running SPEC-OMP2001 benchmarks using the proposed technique on an asymmetric multiprocessor configuration (a=2).

	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 throughput gains for three synthetic benchmarks running in parallel improved with the new scheduler by 1% in the symmetric configuration and by 1.87% in the asymmetric configuration. These results were expected, as

the measured CPU idle time in the symmetric case for three benchmarks was 13.62% (or 12.25% with the proposed scheduler) in comparison to 20% for two applications (or 17.2% with the proposed scheduler). With additional applications running in parallel, the average idle time decreases, leaving less room for improvement for our proposed scheduler.

The jitter for the synthetic benchmarks multiplied by 1000 is shown in Table 8, and was reduced on average by 60% in the symmetric case and by 35% in the asymmetric case. The jitter, measured on five runs of "equake" & "art" as an example, was almost eliminated in the symmetric case and was halved in the asymmetric case. Fairness has improved as well in almost all benchmarks.

Table 8. The average fairness and jitter metrics with the baseline and proposed environments for the synthetic benchmarks and for SPEC-OMP (5 executions of "art" & "equake").

Benchmark	Scheduler	Symmetric		Asymmetric	
		Fairness	Jitter	Fairness	Jitter
Synthetic	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%**
SPEC-OMP	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%**

VIII. Conclusions and Future Work

In this paper, we proposed a new thread assignment mechanism that favors serial phases of applications over parallel phases, in a first attempt to optimize multiple multithreaded applications running in parallel on symmetric and asymmetric multiprocessors. Detailed analysis for any number of multithreaded applications running simultaneously shows potential for improvement in throughput, fairness and jitter metrics. In particular, when two multithreaded scientific applications (SPEC-OMP2001) are run on symmetric as well as on asymmetric multiprocessors, analytical and experimental results show 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 concepts of this work 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. Moreover, the concepts could be used in grid architectures to better exploit the computing power of shared memory nodes by scheduling several multithreaded workloads at once.

There are various architectural implications for this work. First, chip architects designing asymmetric multiprocessors can use the analysis presented in this paper for predicting the effects of asymmetry on various system metrics. We found that as asymmetry between the cores widens, fairness worsens and jitter between execution runtimes increases. Second, exploiting asymmetry requires faster thread migration techniques such as those implemented in this research. We believe that in future designs, hardware may assist the OS in performing these migrations, as opposed to current designs in which the OS migrates threads without any hardware assistance. Third, the performance improvements presented in this paper for asymmetric structures show that asymmetry presents even greater performance potential over symmetric designs than predicted by previous research.

This work also provides insights into a multitude of future research issues in the area of multithreaded application handling in CMP. The analysis 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, with either one active thread or *n* active threads, could be extended to include the whole range from one to *n*. Such extensions could consequently be used to improve system metrics even further, even on current symmetric architectures.

With regard to asymmetric configurations, the analysis 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.

Acknowledgment

The authors would like to thank Dror Feitelson, Idit Keidar, Avi Mendelson and Ronny Ronen for their insightful comments. They would also like to 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.

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

[3] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," in ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992.

[4] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's Law Through EPI Throttling," in Proc. of the 35th ISCA, June 2005.

[5] 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.

[6] 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.

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

[8] A. Fedorova, D. Vengerov, and D. Doucette, "Operating System Scheduling On Heterogeneous Core Systems," in Proc. of OSHMA workshop, 16th PACT, 2007.

[9] D. G. Feitelson and L. Rudolph, “Evaluation of design choices for gang scheduling using distributed hierarchical control,” in J. Parallel & Distributed Computing 35(1), May 1996.

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

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

[12] Mark D. Hill and Michael R. Marty, “Amdahl's Law in the Multicore Era,” in IEEE Computer, July 2008.

[13] 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.

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

[15] 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.

[16] D.S. Nikolopoulos, C.D. Antonopoulos, I.E. Venetis, P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou, “Achieving Multiprogramming Scalability on Intel SMP Platforms: Nanothreading in the Linux Kernel,” in PARCO 1999.

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

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

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

[20] U. Schwiegeishohn, R. Yahyapour, “Improving first-come-first-serve job scheduling by gang scheduling,” in IPPS'98 Workshop, March 1998.

[21] A. Snavely, D. Tullsen, and G. Voelker, “Symbiotic job scheduling with priorities for a simultaneous multithreading processor,” in proc. of the 2002 ACM SIGMETRICS, 2002.

[22] J. Vera, F.J. Cazorla, A. Pajuelo, O.J. Santana, E. Fernández and M. Valero, “FAME: FAirly MEasuring Multithreaded Architectures,” in IEEE-ACM PACT Conference. Brasov, 2007.

[23] I.E. Venetis, D.S. Nikolopoulos, and T.S. Papatheodorou, “A Transparent Operating System Infrastructure for Embedding Adaptability to Thread-Based Programming Models,” in EuroPar 2001.

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