Task Scheduling Based On Thread Essence and Resource Limitations

Tomer Y. Morad, Avinoam Kolodny, and Uri C. Weiser
Department of Electrical Engineering, Technion, Haifa, Israel
Email: {tomerm@tx, kolodny@ee, uri.weiser@ee}.technion.ac.il

***Abstract*—Scheduling of threads based on the concept of thread essence is proposed in this paper. Multithreaded applications contain serial phases (single thread) and parallel phases (many threads). We propose a thread assignment mechanism that takes into account the essence of the threads in simultaneously-running applications that grants higher priority to applications during their critical-serial phases, for environments where there are more threads than cores. Furthermore, our proposed scheduler considers the limited resources of the system by reducing the number of context switches when there are more ready threads than cores. Analytic and experimental evaluation of the proposed thread assignment mechanism on both symmetric and emulated 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%.**

***Index Terms*—Asymmetric Multiprocessors, Operating Systems, Scheduling**

I. INTRODUCTION

Operating system schedulers' most important task is to allocate the available system resources to the workload defined by the users. Workloads consist of a set of threads, each with its own requirements, such as memory bandwidth, IO, floating point unit usage, branch prediction, and dependency on other threads. The resources of the system are comprised of the available processing cores, their type and their performance, the memory hierarchy, volatile and non-volatile memory, network, graphics and more.

The schedulers carry out the scheduling task by assigning resources to threads, and in most cases also revoking resources from threads, as done by operating systems that support preemption. There are various metrics for assessment of a scheduler's performance, such as system throughput, latency, fairness, jitter, and power consumption.

Schedulers must consider the different attributes of threads in order to achieve good scheduling performance. Threads may have user-defined attributes, such as static priorities, target performance, and maximum allowed power consumption. Another class of attributes includes platform-independent attributes, such as the memory footprint, shared data with other threads, types of instructions, memory access pattern, IO access pattern and more. A third class of attributes includes platform-dependent attributes, such as branch prediction confidence level, miss rate, instructions per cycle (IPC) and more. All these attributes of threads are referred to in this paper as the essence of threads.

An example of a platform independent attribute is the "interactivity" metric used by the Linux scheduler [1]. The Linux scheduler classifies threads as interactive, or IO-bound, when their sleep time, which is the time threads voluntarily release the processor, is high. IO-bound threads are granted higher priority by the Linux scheduler, since these threads exhibit long waits between relatively short CPU usage times. Granting priority to IO-bound threads results in lower latency and better performance perceived by the users.

Indeed, there are papers that suggest considering some of the thread attributes when scheduling threads. Knauerhase et al. [13] suggest co-scheduling threads based on cache usage. Zhuravlev et al. [28] suggest co-scheduling threads based on memory bandwidth usage, memory controller contention and contention on the prefetching hardware.

One of the attributes that is useful for scheduling is the inner state of the applications in the context of multithreaded applications. Morad et al. [18] suggest scheduling threads based on their application's phase, whether serial or parallel. All of the above mentioned thread attributes have an effect on the performance of the scheduler.

The scheduler's task is to choose which of the ready threads to run and which resources to grant for each thread. Having many ready threads, therefore, is important for achieving high scheduling performance, since the scheduler will have more possibilities to choose from. Granting of resources to threads, however, must take into account the limited resources available in the system. For example, a scheduler might achieve better performance on a multi-core processor with a workload of many bandwidth-hungry threads, by keeping some of the cores idle, since any additional concurrently running thread will increase the number of misses in the shared cache for all of the threads and will exhaust the available memory bandwidth.

This paper presents an example of a concept: thread scheduling based on the essence of the threads and on the limited resources of the system. In particular, we present a way to maximize the number of ready threads for the scheduler when more than one multithreaded application runs in parallel, by examining the essence of the running applications. Given the higher number of ready threads, our proposed scheduler is able to better choose which and how many threads to run, achieving better performance in various metrics.

II. Multiple Multithreaded Applications

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 [20], 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 [14], 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 the serial phases, only one thread is active, whereas the parallel phases are comprised of many concurrently active threads. The number of threads as a function of time during the execution of the "equake" SPEC-OMP benchmark is shown in Fig. 1. The serial and parallel phases of the "equake" benchmark can be distinguished in Fig. 1 by the number of running threads (only one thread is active in the serial phase, and several threads are active in the parallel phase).

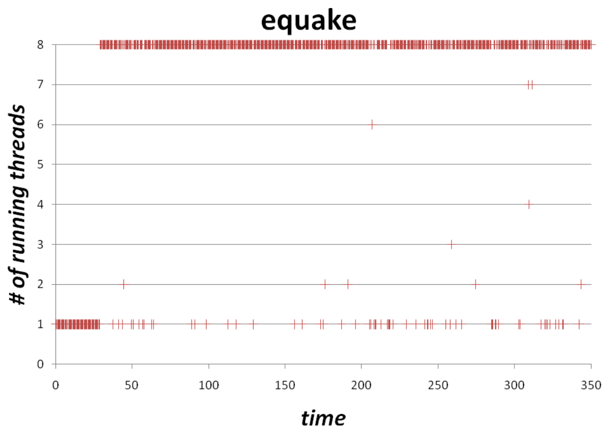


Fig. 1. The number of running threads as a function of time for the "equake" benchmark in the SPEC-OMP suite, on an 8-way multiprocessor. The serial and parallel phases can be distinguished by the number of running threads at each point of time. The sampling frequency in the figure is twice per second.

When several multithreaded applications run simultaneously on a multiprocessor, the serial thread of some applications may be available for execution together with the threads of the other applications. Fig. 2 shows an example of the four possible joint states of two multithreaded applications running simultaneously. The vertical axis represents time, and the number of active threads of each application is shown for each point of 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 may result in lower throughput, jitter in applications' runtimes (unpredictable performance), and unfairness between applications. These undesired characteristics may happen because the serial phases, which are the 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.

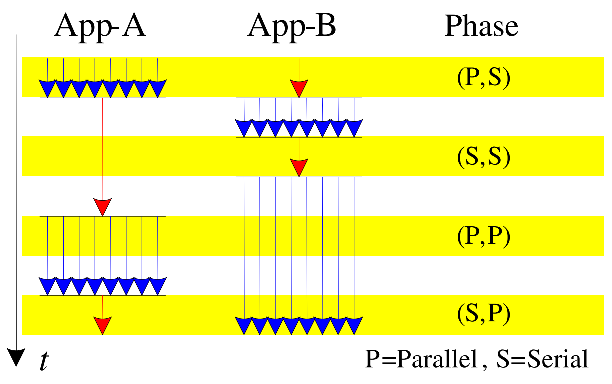


Fig. 2. Illustration of joint states of two sample applications running simultaneously.

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. 3 shows transitions among 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). The large arrows on the state transition arcs denote the most likely transition. The proposed thread assignment technique, shown in Fig. 3b, 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. 3a), 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 improves throughput by reducing the time spent in (S,S) in which there are idle cores, resulting in a greater number of ready threads.

The new thread assignment technique corresponding to the transition likelihoods illustrated in Fig. 3b 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 [12] with a current version of the Linux operating system [1], 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 the faster cores.

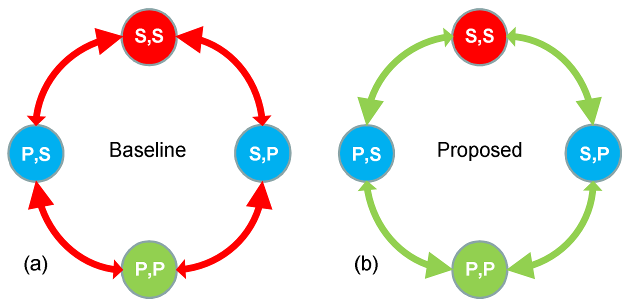


Fig. 3. Illustration of the four possible joint states of two applications running in parallel. The large arrows denote the most likely transition. Fig. 3a illustrates the states and the likely transitions in the baseline scheduler, and Fig. 3b illustrates the states and the likely transitions in the proposed scheduler, where the serial threads are favored over parallel threads.

III. Related Work

There are a number of papers addressing the scheduling of single-threaded applications on asymmetric/heterogeneous multiprocessors [14], which are based on sampling of runtime performance on the different core types. Kumar et al. [15] 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. [27] 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][17]. 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 with a standard Linux scheduler as a baseline environment that does not employ the proposals from previous work, since these proposals are tailored for single multithreaded applications. When more than one multithreaded application is run in parallel, a scheduler with these proposals will perform similarly to the baseline environment, and hence will only add to the complexity of evaluating our proposed techniques without offering additional insights.

Several papers explore fairness and throughput in SMT architectures [10][16][22][24]. We use and extend their throughput and fairness metrics for asymmetric multiprocessors. Other papers explore scheduling multiple multithreaded applications on symmetric systems [3][19][26]. We extend these ideas for asymmetric configurations and present the concept of prioritizing applications based on their phase of execution.

Scheduling threads based on thread attributes has been proposed in previous research [13][18][28]. Knauerhase et al. [13] suggest co-scheduling threads based on cache usage. Zhuravlev et al. [28] suggest co-scheduling threads based on memory bandwidth usage, memory controller contention and contention on the prefetching hardware. We extend these concepts and present the concept of scheduling based on thread essence, while considering the limited resources of the system on which the threads are run.

IV. Emulation Environment

All measurements in this paper were performed on an 8-core multiprocessor (HP ProLiant DL580) [12] consisting of four dual-core 2.66GHz Intel Xeon 7020 processors, 667MHz front side bus, and 8GB of DDR2 memory. SMT was disabled for better emulation of symmetric and asymmetric systems. The operating system used was Linux with kernel version 2.6.18, and is referred to in this paper as the baseline environment.

In order to emulate the asymmetric multiprocessor on the symmetric multiprocessor, the frequency (duty cycle) of the cores was changed, as was done in previous research [6][11][18]. Additionally, the Linux kernel was configured accordingly, using the Linux CPU group property CPU_POWER, to employ cores of different speeds.

The benchmarks we used in this research 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 [21] offers various scheduling options for its parallel constructs. 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 [17], and allows higher core utilization on heterogeneous multiprocessors, by reducing the time threads wait for each other.

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.

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

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.

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. 4.

```
while (time<60 seconds) {
    parallel_iterations = random();
    serial_iterations = parallel_iterations * $ratio_{Serial}$ /($ratio_{parallel}$ * $N_{cores}$);
    in each thread { // fork
        for (i=0;i<parallel_iterations;i++)
            perform_calculation();
        calculated_iterations += parallel_iterations; //shared variable
    } //join
    for (i=0;i<serial_iterations;i++)
        perform_calculation();
    calculated_iterations += serial_iterations;
}
print "performance=",calculated_iterations/(time-start_time)
```

Fig. 4. Pseudo-code of the synthetic benchmark.

V. 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 [25]. 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 multi-core 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, and not simply the IPC.

Measuring the throughput of multiple synthetic benchmarks that are 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. 5, and is similar to the "Last" method described in [25]. 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.

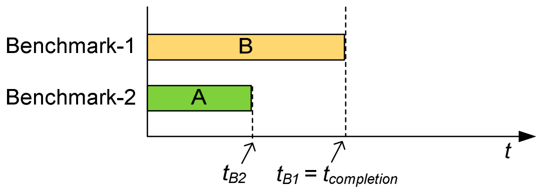


Fig. 5. Example of two multithreaded benchmarks running in parallel. Each application finishes its execution at a different time.

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. 6. 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 [25].

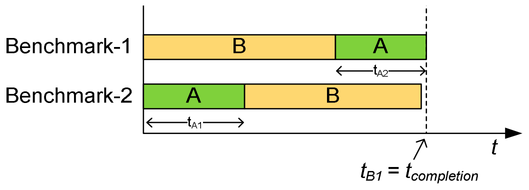


Fig. 6. Two multithreaded benchmarks, each comprising of two applications in a different order. The execution times t_{B1} and t_{B2} of the two benchmarks are similar.

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:

$$speedup_A = \frac{Performance_{A,A\|B}}{Performance_{A,A}} \tag{2}$$

If both applications exhibit the same relative speedup, the system is said to be fair [10][22]. 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:

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

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. 6:

$$Speedup_A = \frac{t_{A,alone}}{\frac{1}{2}\left(t_{A_1} + t_{A_2}\right)} \tag{4}$$

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:

$$Jitter_A = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(\frac{t_{A,n}}{t_{A,avg}} - 1\right)^2} \tag{5}$$

VI. 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 consider an asymmetric multiprocessor with *n* cores: one of the cores is larger as in [17] and faster by a factor (*a*). For simplicity, we assume that the performance factor (*a*) is identical for all workloads. The results in this section can be derived for symmetric multiprocessors by assigning *a*=1.

When only one thread is running in the system, we assume that the scheduler will schedule the thread on the higher performance core. The performance of a serial thread on an idle asymmetric multiprocessor is thus given by:

$$Perf_{serial} = a \tag{6}$$

When a multithreaded application with *n* threads runs on a multiprocessor with *n* cores, we assume that the scheduler will distribute the threads on the cores so that no core will be left idle. This assumption makes sense for sufficiently small values of (*a*), or ($1 \le a < 2$). For higher values of (*a*) the scheduler could potentially achieve better performance by scheduling more threads on the larger core and leave some of the smaller cores idle. For simplicity, we assume that (*a*) is sufficiently small ($1 \le a < 2$) in our analysis. The performance of a parallel application on an idle asymmetric multiprocessor is thus given by:

$$Perf_{parallel} = n - 1 + a \tag{7}$$

We consider the case where two concurrently running applications, App_A and App_B, have *n* active threads in their parallel phases.

When both applications are serial, the scheduler has two options for scheduling. In the first option, one application is scheduled to run on the larger core, and the other application is scheduled to run on a small core. In the second option both applications can run on the larger core. Since we assume that (*a*) is sufficiently small ($1 \le a < 2$) this option is not considered in our analysis. The maximum speedup in this case will be achieved when the serial application runs on the larger core:

$$Speedup_{(S,S),\max} = 1 \tag{8}$$

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

$$Speedup_{(S,S),\min} = \frac{1}{a} \tag{9}$$

Given that each application has an equal chance to run on the larger core, the speedup is as follows:

$$Speedup_{(S,S),avg} = \frac{1}{2}\cdot\frac{1}{a}+\frac{1}{2}\cdot 1 = \frac{1}{2}\left(1+\frac{1}{a}\right) \tag{10}$$

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.

The maximum speedup for a parallel application will therefore be achieved when two threads of the parallel application are scheduled on the larger core, and each of the other threads of the parallel application run in conjunction with another thread on a small core:

$$Speedup_{(P,P),\max} = \frac{(n-2)\cdot\frac{1}{2}+2\cdot\frac{a}{2}}{n-1+a} = \frac{n-2+2a}{2(n-1+a)} \tag{11}$$

The minimum speedup for a parallel application will be achieved when each of the threads of the parallel application runs in conjunction with another thread on a small core, and none of the threads run on the faster core:

$$Speedup_{(P,P),\min} = \frac{(n)\cdot\frac{1}{2}+0\cdot\frac{a}{2}}{n-1+a} = \frac{n}{2(n-1+a)} \tag{12}$$

In the average case, each of the parallel applications will exhibit a speedup of 0.5:

$$Speedup_{(P,P),avg} = \left(\frac{n}{2n}\frac{n-1}{2n-1}\right)\frac{(n-2)\cdot\frac{1}{2}+2\cdot\frac{a}{2}}{n-1+a} + \left(2\frac{n}{2n}\frac{n}{2n-1}\right)\frac{(n-1)\cdot\frac{1}{2}+1\cdot\frac{a}{2}}{n-1+a} + \left(\frac{n}{2n}\frac{n-1}{2n-1}\right)\frac{(n-2)\cdot\frac{1}{2}+2\cdot\frac{a}{2}}{n-1+a} = \frac{1}{2} \tag{13}$$

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 on a small core:

$$Speedup_{(S,P),\min} = \frac{1}{2}\cdot\frac{1}{a} = \frac{1}{2a} \tag{14}$$

In the best case for the serial application, it will exhibit no slowdown as it will be scheduled to run on the large core by itself:

$$Speedup_{(S,P),\max} = 1 \tag{15}$$

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 larger core is n^{-1}. The average performance of the serial thread when running concurrently with a parallel application is thus:

$$Speedup_{(S,P),avg} = \frac{2}{n+1}\left(\frac{1}{n}\cdot\frac{1}{2}+\frac{n-1}{n}\cdot\frac{1}{2a}\right)+ \frac{n-1}{n+1}\left(\frac{1}{n}\cdot 1+\frac{n-1}{n}\cdot\frac{1}{a}\right) = \frac{n-1+a}{a(n+1)} \tag{16}$$

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

$$Speedup_{(P,S),\max} = \frac{1\cdot\frac{1}{2}+1\cdot a+(n-2)\cdot 1}{n-1+a} = \frac{n-\frac{3}{2}+a}{n-1+a} \tag{17}$$

The minimum speedup is achieved when the serial thread is scheduled alone on the larger core:

$$Speedup_{(P,S),\min} = \frac{(n-2)\cdot 1+2\cdot\frac{1}{2}+0\cdot a}{n-1+a} = \frac{n-1}{n-1+a} \tag{18}$$

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

$$Perf_{(P,S),avg} = \frac{1}{n-1+a} \cdot \left(\frac{2}{n+1}\left(\frac{1}{n}\cdot(n-1+\frac{a}{2}) + \frac{n-1}{n}\cdot(n-\frac{3}{2}+a)\right) + \frac{n-1}{n+1}\left(\frac{1}{n}\cdot(n-1) + \frac{n-1}{n}\cdot(n-2+a)\right) \right) = \frac{n}{n+1} \quad (19)$$

The minimum, maximum and average speedups as calculated in the above equations are summarized in Table 1 for asymmetric multiprocessors.

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)	$\frac{n}{2(n-1+a)}$	$\frac{1}{2}$	$\frac{n-2+2a}{2(n-1+a)}$	$\frac{n-2+2a}{n}$

Table 2 shows the speedups calculated for symmetric multiprocessors (a=1).

TABLE 2. SPEEDUPS (MIN, AVERAGE, MAX) FOR APPLICATION "A" IN THE BASELINE ENVIRONMENT ON THE SYMMETRIC MULTIPROCESSOR. n=NUMBER OF CORES.

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

Fairness is calculated according to (3). When two applications are both in their serial phase, the lower bound for fairness is given by dividing the minimum and maximum speedups of the state (S,S):

$$Fairness_{(S,S)} = \frac{Speedup_{(S,S),\min}}{Speedup_{(S,S),\max}} = \frac{1}{a} \quad (20)$$

Similarly, the lower bound for fairness when both applications are in their parallel phases is given by dividing the minimum and maximum speedups of the state (P,P):

$$Fairness_{(P,P)} = \frac{n}{n-2+2a} \quad (21)$$

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):

$$Fairness_{(S,P),1} = \min\left(\frac{Speedup_{(S,P),\min}}{Speedup_{(P,S),\max}}, \frac{Speedup_{(P,S),\max}}{Speedup_{(S,P),\min}}\right) = \frac{n-1+a}{2a\left(n-3/2+a\right)} \quad (22)$$

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):

$$Fairness_{(S,P),2} = \min\left(\frac{Speedup_{(S,P),\max}}{Speedup_{(P,S),\min}}, \frac{Speedup_{(P,S),\min}}{Speedup_{(S,P),\max}}\right) = \frac{n-1}{n-1+a} \quad (23)$$

The lower bounds of the fairness metric as calculated in the above equations for the baseline environment are summarized in Table 3.

TABLE 3. LOWER BOUND FOR FAIRNESS IN THE BASELINE ENVIRONMENT.

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

The minimum and maximum speedups in the different joint phases, as detailed in Table 1, and the equations for the lower bound of fairness, as detailed in Table 3, 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 the runtimes increases.

We extend the analysis in this section for $k>2$ multithreaded applications running in parallel. The extension results are detailed in Table 4. 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 4. 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	$\frac{n}{k(n-1+a)}$	$\frac{n-1+a-\frac{k-1}{2}}{n-1+a}$	$k\frac{n-1+a-\frac{k-1}{2}}{n}$

In this section we presented analytic tools to predict the speedup and fairness of applications running in parallel. The analysis predicts that applications may exhibit significant slowdowns due to the contentions with other running applications. One of the factors shown to affect the slowdowns is the joint-phases of the applications that are currently running. These joint-phases are hard to predict in advance, potentially resulting in unpredictable performance of the applications. Moreover, each application may exhibit different slowdowns, as shown in the lower bounds of the fairness equations. The analysis also predicts that as the asymmetry widens, the possible jitter increases and the lower bound of fairness decreases.

VII. 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 examines the essence of the running applications and 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 5, which differs from Table 1 in states (S,P) and (P,S).

TABLE 5. 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 number of ready threads will be greater on average with the proposed scheduler, since the serial threads will execute faster and allow the parallel phases to start executing sooner. The proposed scheduler will thus be able to increase the utilization of the multiprocessor by scheduling additional threads that will run concurrently.

In state (S,S) on the asymmetric multiprocessor, there are two active serial threads but only one of them is granted the larger 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 processing 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 detailed in Table 6.

The Linux scheduler was extended to support the proposed algorithm. The proposed scheduler continuously monitors the number of ready threads in each thread group, and hence can detect whether an application is in its parallel phase or in its serial phase. This is performed in O(1) time whenever a thread changes its ready state. In our implementation, 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

OpenMP application frequently switched between one and two active threads.

TABLE 6. 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	$\frac{n}{k\left(n-1+a\right)}$	$\frac{n-k+1}{n-1+a}$	$k\frac{n-k+1}{n}$

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 for maximizing the number of ready threads, results in many unnecessary context switches that thrash the cache and lower the overall throughput of the system.

Our proposed technique strives to avoid this situation by considering the limited resources of the system, and 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][23]. 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 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 are granted more priority. In asymmetric configurations, the high priority given to these threads will usually result in migration to the higher performance core.

The asymmetric multiprocessor is emulated by changing the frequency (duty cycle) of seven out of eight cores in our symmetric multiprocessor, as was 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 larger 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.

VIII. 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 7 for the asymmetric configuration.

TABLE 7. MEASURED SPEEDUP OF TWO CONCURRENTLY RUNNING SYNTHETIC BENCHMARKS USING THE PROPOSED TECHNIQUE ON AN ASYMMETRIC MULTIPROCESSOR CONFIGURATION (a=2).

	(∞:1)	**(8:1)**	**(4:1)**	**(2:1)**	**(1:1)**	**(1:2)**	**(1:4)**	**(1:8)**	**(1:∞)**
(∞:1)	1%								
(8:1)	-1%	1%							
(4:1)	-1%	1%	1%						
(2:1)	0%	-1%	4%	4%					
(1:1)	-2%	1%	3%	4%	7%				
(1:2)	0%	1%	3%	5%	8%	7%			
(1:4)	-2%	2%	0%	6%	9%	11%	8%		
(1:8)	-2%	1%	3%	8%	7%	15%	18%	3%	
(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. 7 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.

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.

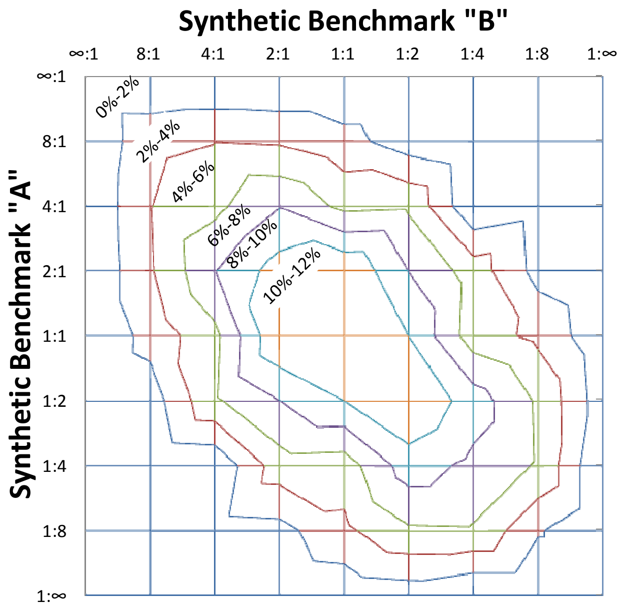


Fig. 7. Experimental contours of the speedup of two concurrently running synthetic benchmarks when the proposed technique is used on a symmetric multiprocessor configuration (*a*=1).

Table 8 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. 6. 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 8. MEASURED SPEEDUP OF TWO CONCURRENTLY RUNNING SPEC-OMP2001 BENCHMARKS USING THE PROPOSED TECHNIQUE ON AN ASYMMETRIC MULTIPROCESSOR CONFIGURATION (*a*=2).

	wup	swi	mgr	app	equ	aps	gaf	fma	art	amm
wupwise	2%									
swim	8%	1%								
mgrid	4%	4%	4%							
applu	2%	-2%	3%	1%						
equake	3%	0%	4%	0%	0%					
apsi	12%	15%	7%	12%	9%	16%				
gafort	1%	3%	2%	-2%	2%	7%	0%			
fma3d	1%	5%	3%	-3%	0%	9%	3%	3%		
art	2%	0%	4%	3%	-1%	15%	-1%	2%	-1%	
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%										

When running two SPEC-OMP benchmarks concurrently as in Fig. 6, compared with running the benchmarks sequentially only one application at a time, throughput increases by 36% on average in the baseline scheduler, and by 38% on average on the proposed scheduler. In some cases, however, the "ammp" and "wupwise" benchmarks ran faster sequentially in the baseline and proposed schedulers compared to running concurrently. This slowdown was caused by the contention between the threads on the system resources. A scheduler that is aware of the system resource limits could potentially mitigate this slowdown by running only the threads that the system has enough resources for them to run.

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 9, 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" and "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. Notably, the lower bound of fairness as measured in five runs of "equake" and "art" improved by 26%.

TABLE 9. 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 (A=1)		Asymmetric (A=2)	
		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%**

IX. CONCLUSIONS AND FUTURE WORK

In multiprocessors running multiple multithreaded applications, thread scheduling may be performed based on the essence of the threads and on the resource limitations of the system. As an example, our proposed thread assignment mechanism examines the essence of the threads and 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 several multithreaded applications running simultaneously shows potential for improvements in throughput, fairness and the 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 today's 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. A first step towards faster migration techniques was presented in this research, and future work should continue to explore how to react fast to the constant and continuous changes in the available system resources. 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.

Future work can extend the analysis 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. The analysis could also be extended to support various configurations of asymmetric multiprocessors, such as more than two types of cores. Furthermore, the analysis could take into account different speedups for different applications on each core type.

This work also provides insights into a multitude of future research issues in the area of multithreaded application handling in CMP. Essence of threads can be further explored to point out the most influential attributes that affect scheduling. Such attributes may include cache usage, memory bandwidth, dependencies on other threads, IO access patterns and more.

Scheduling according to resource limitations should be explored further. We have shown that in some cases it is better to run multithreaded applications sequentially than to run them concurrently. Future schedulers should avoid scheduling threads if they will use more resources than are currently available in the system. This will allow better usage of the system resources while conserving power.

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," in 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] HP Technical Document DA-12195, "HP Proliant DL580 Generation 3 (G3)," Version 30 – May 2007: http://h18004.www1.hp.com/products/quickspecs/12195_div/12195_div.pdf.

[13] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," in IEEE Micro 28, 2008.

[14] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," in IEEE Computer, July 2008.

[15] 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.

[16] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," in Proc. of the ISPASS, pages 164–171, 2001.

[17] 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.

[18] T.Y. Morad, A. Kolodny, and U.C. Weiser, "Scheduling Multiple Multithreaded Applications on Asymmetric and Symmetric Chip Multiprocessors," in the 3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP'10), Dalian, 2010.

[19] 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.

[20] K. Olukotun and L. Hammond, "The Future of Microprocessors," in ACM Queue, Vol. 3, No. 7, 2005.

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

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

[23] U. Schwiegeishohn, R. Yahyapour, "Improving First-Come-First-Serve Job Scheduling by Gang Scheduling," in IPPS'98 Workshop, March 1998.

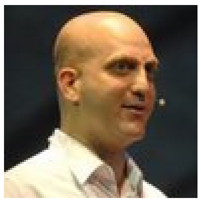
[24] A. Snavely, D. Tullsen, and G. Voelker, "Symbiotic job scheduling with priorities for a simultaneous multithreading processor," in Proc. of the 2002 ACM SIGMETRICS, 2002.

[25] 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.

[26] 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.

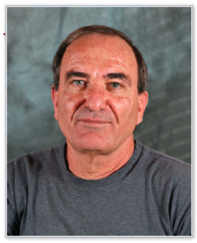
[27] J.A. Winter and D.H. Albonesi, "Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures," in Proc. of the 38th DSN, June 2008.

[28] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in Proc. of ASPLOS'10, New York, 2010.



Tomer Y. Morad is a Ph.D. Candidate at the Technion – Israel Institute of Technology. He is also the Chief Technology Officer in transSpot Ltd. Tomer earned his B.Sc. and M.Sc. in Electrical Engineering from the Technion in 2001 and 2005 respectively. His main research interests are asymmetric cluster chip multiprocessors and operating system schedulers.

Avinoam Kolodny is an associate professor of electrical engineering at the Technion – Israel Institute of Technology. He joined Intel after completing his doctorate in microelectronics at the Technion in 1980. During twenty years with the company he was engaged in diverse areas including non-volatile memory device physics, electronic design automation and organizational development. He pioneered static timing analysis of processors as the lead developer of the CLCD tool, served as Intel's corporate CAD system architect in California during the co-development of the RLS system and the 486 processor, and was manager of Intel's performance verification CAD group in Israel. He has been a member of the Faculty of Electrical Engineering at the Technion since 2000. His current research is focused primarily on interconnect issues in VLSI systems, covering all levels from physical design of wires to networks on chip and multi-core systems.



Uri C. Weiser is a visiting professor at the Electrical Engineering department of the Technion – Israel Institute of Technology. He is also active on the advisory boards of numerous startups. He earned his Ph.D. in CS from the University of Utah, Salt Lake City. Uri worked at Intel from 1988–2006 where he initiated and drove the definition of the first Pentium® processor, led the Intel's MMX™ technology, co-invented the Trace Cache, co-managed Intel's new Design Center at Austin, Texas and formed an advanced media applications research activity. Uri was appointed Intel Fellow; he is an ACM Fellow, and Fellow of the IEEE. Prior to his career at Intel, Uri C. Weiser worked at the Israeli Department of Defense and later with National Semiconductor Design Center in Israel, where he led the design of the NS32532 microprocessor.