J. Parallel Distrib. Comput. 95 (2016) 3–14

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

EFS: Energy-Friendly Scheduler for memory bandwidth constrained systems

Tomer Y. Morad [a,b,*], Noam Shalev [a], Idit Keidar [a], Avinoam Kolodny [a], Uri C. Weiser [a]

[a] Viterbi Faculty of Electrical Engineering, Technion, Haifa, Israel
[b] Jacobs Technion-Cornell Institute, Cornell Tech, New York, NY, USA

HIGHLIGHTS

- We develop a simple model that predicts throughput and power of concurrently running threads in the presence of interference.
- We propose an operating system scheduler, EFS, that minimizes energy per work by preventing ineffective utilization of system resources.
- We implement EFS which achieves up to 32% reduction in system energy as measured by an external power meter.

ARTICLE INFO

Article history:
Received 4 September 2015
Received in revised form
5 February 2016
Accepted 22 March 2016
Available online 19 April 2016

Keywords:
Energy Friendly Scheduler
Scheduling
Energy efficiency
Performance monitors
Multicores

ABSTRACT

Additional transistors available in each process generation are used to increase the number of cores on chip. This trend results in high execution unit performance relative to other available resources, such as memory bandwidth, I/O bandwidth, and power. Consequently, the performance bottleneck in modern systems has shifted from the execution units to other resources. In this paper we propose a dynamic scheduling scheme that avoids bottlenecks and thus saves energy. Current operating system schedulers are designed to always assign threads to available cores. We show that this approach may result in excessive loads on other resources, which can ultimately hamper performance and waste energy. Thus, perhaps paradoxically, in some cases it may be advantageous to under-utilize on-chip computing resources in order to achieve better performance and energy efficiency. More generally, we argue that operating system schedulers should consider multiple resources, such as memory bandwidth, dynamic cache conflicts, and I/O bandwidth. We develop this concept in the context of memory bandwidth, which is a critical bottleneck in many systems. To this end, we suggest a model that predicts threads' throughput and power consumption based on contention on the memory bus. We use this model to design EFS (Energy-Friendly Scheduler), a new energy-efficient scheduler, which schedules new threads only when the benefit of the added throughput outweighs the cost of powering up additional cores. The idea is simple, and we implement it in Linux using performance monitors readily available in current microprocessors. Execution results on a real multicore system with EFS show up to 32% energy reductions in resource-constrained SPEC-CPU2006 benchmarks, as measured using an external power meter.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Multicore architectures have become the de-facto choice for general-purpose computing. Recent trends show that in each process generation, the number of cores increases twofold, outpacing the increase in other system resources. Emerging architectures, therefore, offer an increasing ratio of raw execution power relative to other system resources, such as memory bandwidth, I/O bandwidth, cache capacity, and power. This has caused performance bottlenecks to shift away from execution units. We make the case therefore that in today's systems, the traditional approach of aggressively scheduling threads on all available cores can stress other resources. Increased resource contention, in turn, degrades performance and wastes energy. While our concepts readily apply to many shared resources, such as memory capacity, network bandwidth, storage bandwidth, and a power envelope, we focus here on memory bandwidth, which emerges as a chief bottleneck in modern systems. We target building a new class of schedulers that save energy by using only a subset of the available cores when a bottleneck forms on other system resources.

* Corresponding author.
E-mail addresses: tomerm@tx.technion.ac.il (T.Y. Morad), noams@tx.technion.ac.il (N. Shalev), idish@ee.technion.ac.il (I. Keidar), kolodny@ee.technion.ac.il (A. Kolodny), uri.weiser@ee.technion.ac.il (U.C. Weiser).

http://dx.doi.org/10.1016/j.jpdc.2016.03.007
0743-7315/© 2016 Elsevier Inc. All rights reserved.

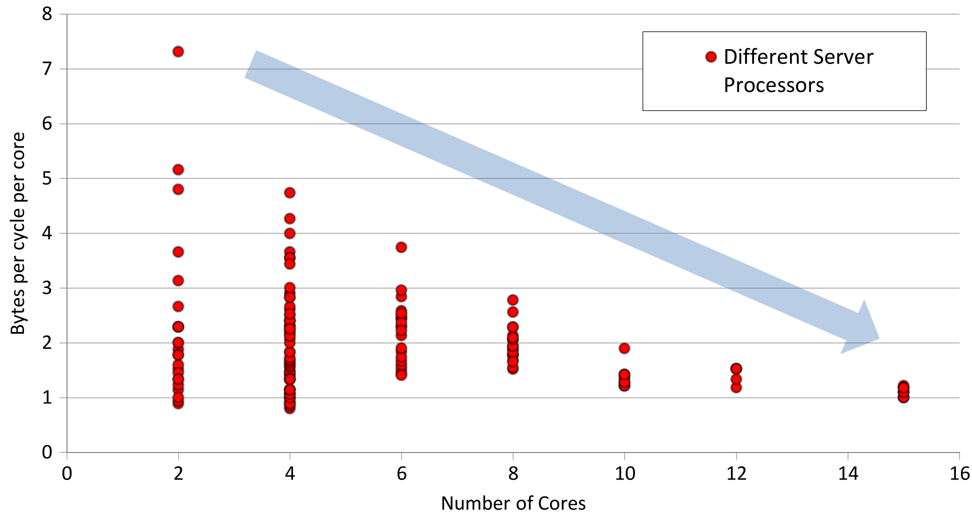


Fig. 1. Maximum quoted bandwidth to external memory for recent server processors [18]. Each data point represents a different chip product.

Indeed, in each process generation, the memory bandwidth available for the entire chip increases only mildly, and is outpaced by the increase in execution capacity. The maximum quoted memory bandwidths of recent server processors [18] and the number of cores are shown in Fig. 1. The figure shows that the available bandwidth per core decreases as more cores are introduced. For example, current dual-core processors (such as Intel E5-2637) offer 7.3 bytes per cycle per core whereas today's 15-core processors (e.g., Intel E7-8890v2) offer only 1 byte per cycle per core. The International Technology Roadmap for Semiconductors (ITRS) expected trends [19] further indicate that from 2015 to 2020, the number of transistors in state-of-the art processors is expected to triple, the pin-count is expected to increase by 27%, whereas the memory bus frequency will remain roughly the same, effectively more than halving the available bandwidth per transistor.

Prior research has proposed to deal with resource contention in a variety of ways, such as co-scheduling threads that least interfere with each other [29,49,50] or partitioning system resources to provide quality of service [2,11,14,15,20,25,38] (see Section 2 for more details). However, these works do not focus on energy efficiency. Thread packing heuristics [42,45] target energy efficiency, but these works do not directly measure or model resource contention, nor are these works applicable to multi-programmed workloads (independent programs in parallel). In this paper, we advocate enforcing a limit on usage of the bottleneck resource at the cost of leaving other resources (most notably, cores) idle. We build an *Energy-Friendly Scheduler* (EFS), which uses this approach in order to conserve energy and sometimes also improve performance.

When multiple threads contend on a shared resource, they are subject to *collisions*, which cause stalls and waste energy. Consider for example two threads attempting to access the shared memory bus simultaneously. In this case the memory controller will grant one thread access to the bus, while the other thread will be stalled. Not only is the stalled thread's operation slowed down, but its core also typically waits at a high power state, since dynamic voltage scaling is unjustified at such a short time interval. Collisions are corroborated by a second effect, namely *destructive interference*, which occurs when different programs access memory locations mapped to the same cache line. Such an access pattern may incur many cache misses as each program evicts the other's useful cached data. In Section 3 we suggest an on-line model for predicting throughput and energy consumption when collisions are present. We show that our model accurately predicts throughput when there is no destructive interference.

In Section 4, we propose a dynamic scheduling scheme that saves energy by minimizing collisions. The key idea is to avoid overloading bottleneck resources. This is done by predicting resource usage of running threads and preventing execution of threads that may overload bottlenecks.

In Section 5 we evaluate our proof-of-concept implementation of EFS on a multicore system. EFS reduces energy consumption by up to 32% and by an average of 7.7% when running several instances of the same SPEC-CPU2006 benchmarks on a real system as measured using an external power meter.

In summary, our contributions in this paper are as follows:

- We develop a simple model that predicts throughput and power of concurrently running threads in the presence of interference.
- We propose an operating system scheduler, EFS, that minimizes energy per work by preventing ineffective utilization of system resources.
- We implement EFS which achieves up to 32% reduction in system energy as measured by an external power meter.

2. Related work

We target building a new class of schedulers that save energy by using only a subset of the available cores when a bottleneck forms on other system resources. While many previous works have strived to optimize resource usage in CMPs, our work is the first that we know of to enforce a limit on resource usage while leaving other resources (most notably, cores) idle.

The idea to take into account resource consumption, and in particular, bandwidth usage, in selecting threads to schedule is not new; see the papers surveyed in [50] as well as [3–5,9,21,22,24,26,27,29,33,34,39,36,41,49]. However, to the best of our knowledge, all previous works in this vein focused on identifying sets of threads that interfere minimally with each other in order to co-schedule them. Unlike our work, they did not refrain from scheduling a collection of threads that exceed resource limitations, but rather always attempted to keep all cores busy. Perhaps this approach is rooted in the traditional importance attributed to full utilization of compute resources. Our results show that, contrary to this belief, sometimes leaving cores idle can improve performance and significantly reduce energy consumption.

Several papers suggest that the runtime environments choose the number of threads of multithreaded programs to spawn at runtime [9,13,30]. EFS, on the other hand, addresses multi-programmed workloads and does not require altering the applications' binaries or runtime libraries.

Other papers propose to perform core throttling in order to achieve fairness [8,47]. Although this approach may cause cores to stall at a fine grain, unlike our work, they do not refrain from keeping all cores busy, which ultimately results in inefficiencies due to exceeding resource limitations.

Colmenares et al. [6] propose a new operating system that pre-allocates system resources to threads in order to ensure fairness and guarantee performance. This requires threads to explicitly acquire and release system resources. EFS, on the other hand, allows dynamic allocation of system resources with minimal changes to the operating system and with no changes to application binaries.

Thread packing for multithreaded workloads [42,45] may lead to leaving several cores idle by scheduling more threads on fewer cores. Our method differs from these prior works by targeting multi-programmed workloads and by using a finer grained control knob for selecting the subset of threads to run together. Moreover, our task selection is based on resource usage, whereas these works [42,45] do not take into account the resource requirements of the running threads.

Several papers partition available hardware among applications in order to provide Quality of Service (QoS) [2,11,14,15,20,25,38]. For example, Guo et al. [11] explore a shared cache management framework for CMP architectures, where jobs whose cache requirements cannot be met are rejected. These works, however, rely on static user-defined QoS levels. EFS, on the other hand, operates dynamically and does not require user-defined QoS levels. Cache partitioning, e.g., as presented in [11], is in fact orthogonal to our approach, and can be combined with it.

In cloud computing, tasks are distributed among servers using admission control. Several papers and commercial systems [7,32,31,48] have studied how to select the tasks to schedule, considering their resource demands. This requires a priori knowledge of the tasks, as well as a cluster-level scheduler (or dispatcher). EFS works at a much finer granularity—it can detect threads' requirements during different program phases and react to them quickly. It can be used in conjunction with cluster-level schedulers to further improve energy efficiency. Moreover, when cluster-level schedulers are not present, such as in personal devices, a resource-aware scheduler as we suggest is critical for energy efficiency. Note further that EFS can run without any a priori information of the workloads.

Prior research has estimated the bandwidth utilized by several threads that run together using a simple additive model, where the bandwidth of two threads is the sum of their solo bandwidths (e.g., Section 4 in [1]). The additive model's main limitation is that it does not account for collisions, neither when estimating the solo bandwidth of a thread when measured under contention nor when estimating the bandwidth of a set of arbitrary threads whose solo bandwidth requirements are known. Our proposed model, on the other hand, takes collisions into account to provide more accurate estimates. Several papers preset models for collisions on the memory bus [23,44]. Our work presents the first model for predicting collisions of threads with different bandwidth requirements, which can be calculated on-line by a scheduler. It is also the first work that we know of to make use of a model for thread contention on the memory bus, provide an end-to-end scheduler implementation that prevents bottlenecks by leaving cores idle and to demonstrate energy savings on a real system.

Thread scheduling algorithms for uniprocessors have been researched extensively [10], targeting optimal time allocation to threads in one processing unit. These algorithms cannot be directly applied for scheduling the usage of the memory bus, since unlike in uniprocessors, more than one thread can use the resource (memory bus) at the same time, and the parallel usage of the resource causes interference among threads. EFS uses a model that considers these effects in order to support better scheduling decisions.

3. Energy efficiency under interference

In the absence of interference among threads, it is more energy efficient to run threads in parallel rather than serially on a multicore platform. This is because the energy consumed by external shared resources, such as the power supply fan and disk, is amortized among more threads. When there is interference, on the other hand, it may be more efficient to run threads serially, depending on the level of interference. In Section 3.1 we develop models for throughput and power in the presence of interference on the memory bus. In Section 3.2 we evaluate the throughput model on a real system.

3.1. Throughput and power models

We model the impact of the bandwidth from the processor to the main memory on energy-efficiency and performance. The effect of collisions is observed when multiple threads incur a cache miss at the same time. The memory controller queues these misses and serves them according to its scheduling policy. As a result, all but one of the contending threads are stalled. The collision effect is illustrated in Fig. 2. Since the waiting time is short and unpredictable, stalled cores do not transition into an energy saving state, and thus continue to consume power. Additionally, stalls increase the total execution time of the waiting tasks, and therefore more energy is required to power up other system resources, such as the power supply fan, disk, and on-chip shared resources.

We now develop a model for the interaction among threads due to collisions on a shared resource. We first consider a small average fragment of a program consisting of one memory operation, the latency of the memory access, and computation instructions, as illustrated in Fig. 3. When running alone on the multicore, the execution time of the fragment of thread i is denoted as t_i^{solo}, and it comprises of compute time and stall time:

$$t_i^{solo} = latency_i^{solo} + compute_i. \quad (1)$$

Thread i's average solo bus utilization is the ratio between the average memory latency of the thread when running solo and the execution time of the average program fragment:

$$b_i = \frac{latency_i^{solo}}{t_i^{solo}}. \quad (2)$$

We next consider the same fragment running in parallel with other threads; the latency in parallel execution increases and is denoted by $latency_i^{parallel}$. The execution time of the program fragment in parallel execution is thus given by:

$$t_i^{parallel} = latency_i^{parallel} + compute_i. \quad (3)$$

The average bus utilization of thread i when running in parallel with others is calculated by multiplying the solo utilization by the ratio of the execution times of the same average program fragment in the solo and parallel executions:

$$u_i = b_i \frac{t_i^{solo}}{t_i^{parallel}}. \quad (4)$$

Note that the bus utilization of a thread when running in parallel to other threads is lower than its utilization when running solo, since it performs the same amount of work over a longer period of time. The total bus utilization is given by the following:

$$U = \sum_k u_k. \quad (5)$$

We denote by μ the bus bandwidth in terms of accesses per second. The latency of a memory operation when the thread runs

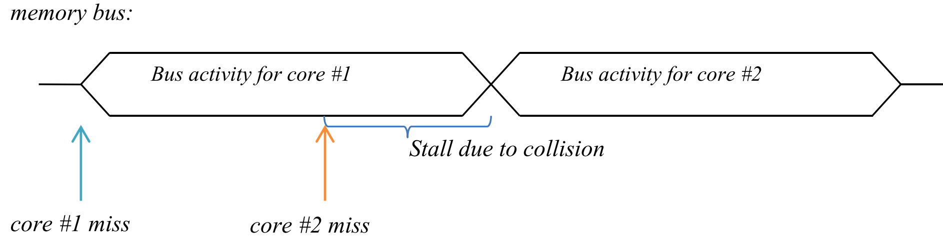


Fig. 2. Example of a stall caused by a collision on the memory bus. Core #2 is stalled since the bus is utilized when it issues a cache miss. During the stall time, core #2 continues to consume power.

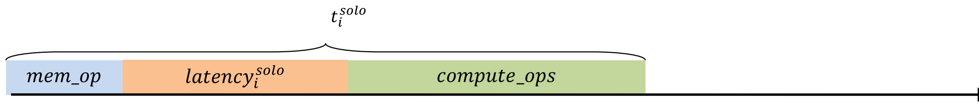


Fig. 3. Average program fragment with one memory operation, bus latency and computation instructions.

solo in the multicore is given by the service rate:

$$latency_i^{solo} = \frac{1}{\mu}. \quad (6)$$

We estimate the memory latency of thread i when running in parallel to other threads by considering the probability ρ_i that it finds the bus busy when it needs it, similar to the way latency is calculated in queueing systems [28]:

$$latency_i^{parallel} = \frac{1}{\mu}(1-\rho_i)\sum_{k=1}^{\infty} k\rho_i^{k-1} = \frac{1}{\mu}\left(\frac{1}{1-\rho_i}\right). \quad (7)$$

The probability of thread i to find the bus busy is given by the following:

$$\rho_i = \sum_{k\neq i} u_k = U - u_i. \quad (8)$$

Using the above equations we derive the solo bus utilization of a thread:

$$b_i = \frac{u_i\,(u_i + 1 - U)}{u_i\,(u_i + 1 - U) + 1 - U}. \quad (9)$$

Note that Eq. (9) offers a way to compute a thread's solo utilization b_i, by measuring U and u_i while other threads interfere. As we show below in our experiments (Fig. 6), the solo utilization can be estimated quite effectively using this method even when other threads interfere.

We now turn to predict how n threads interact given their solo utilizations $b_1 \dots b_n$. Using Eq. (9) we derive a set of n equations:

$$\left\{u_i^2\left(1-\frac{1}{b_i}\right) + u_i\,(1-U)\left(1-\frac{1}{b_i}\right) + 1 - U = 0\right\}_{i=1}^{n}. \quad (10)$$

If the solo utilizations $b_1 \dots b_n$ are known, Eqs. (10) can be solved numerically with $u_1 \dots u_n$ as unknowns, using methods such as the Multivariate Newton's Method, which involves calculating the inverse of the Jacobian matrix and running several iterations until convergence. Solving the equations allows us to predict how threads with different requirements from the memory bus would interact with each other.

When threads run in parallel, they interfere and slow each other down. The slowdown is given by the following:

$$Slowdown_i = \frac{t_i^{solo}}{t_i^{parallel}} = \frac{u_i}{b_i}. \quad (11)$$

We define the parallel throughput of the multicore as the sum of slowdowns multiplied by a coefficient α to adjust the units to work per unit time:

$$Throughput = \alpha \sum_k Slowdown_k. \quad (12)$$

For example, if four threads run in parallel and their execution takes twice their solo times, the slowdowns are 0.5 and the system provides a throughput of 2. If there is no interference, all slowdowns are 1 and the throughput is equal to 4, which is the degree of parallelism.

We now model power consumption of a multicore system. We propose a simple but adequate empirical additive model for power. We have found that the power depends on the number of active cores n in the system, almost regardless of the running programs:

$$Power\,(n) = \sum_{k=0}^{n} P_k. \quad (13)$$

As we show below in our experiments, this simple model provides reasonably accurate predictions of power for our purposes. Note that P_0 accounts for the power dissipated by the system when no core is active, and includes the power supply, disks, and may even include dynamic data center power overheads amortized for the current server.

We focus in this work on minimizing the system energy required to run the program fragments discussed above:

$$Energy = \frac{Power}{Throughput}. \quad (14)$$

We thus use the following expression to evaluate whether the energy per work of executing a current set of threads can be reduced if we run a modified set of threads instead:

$$Energy^{(new)} < Energy^{(current)}. \quad (15)$$

Note that it is easy to target other goals by altering Eq. (15), for example, to an energy-delay goal.

Since the scheduler runs independently on each core, scheduling decisions pertain to deciding if and which thread to schedule in addition to the set of threads that are currently running on other cores. In order to expedite the calculations, instead of calculating the expected throughput for each possible scheduling alternative, we first estimate the highest possible solo bandwidth of an additional thread that would still be energy efficient for the system to run, and then compare the candidate threads' bandwidth estimates to this value. The former is done by solving for equality in Eq. (15):

$$\sum_{k=1}^{n}\frac{u_k}{b_k} - Throughput^{(current)}\frac{Power^{(new)}}{Power^{(current)}} = 0. \quad (16)$$

Eqs. (10) and (16) form a set of $n+1$ equations with $n+1$ variables: $u_1 \dots u_n$ and the solo bandwidth b_n. The first $n-1$

utilizations $u_1 \dots u_{n-1}$ belong to the currently running threads on other cores, whereas u_n and b_n pertain to an additional hypothetical thread that satisfies Eq. (16). Solving the equations results in a value of b_n that can be used as a threshold for candidate threads. If the solo bandwidth of a candidate thread is below the threshold, the system should run it. Otherwise, it would be inefficient to schedule that thread.

Note that our memory bus utilization model suffers from three limitations. The first is lack of modeling of destructive interference, which causes the model to be optimistic. Destructive interference can have an adverse effect on energy and performance. Guz et al. [12] have researched this effect and have shown that there is a performance dip (called "valley" therein), when additional threads are added, due to destructive interference. While prior research has explored ways to model destructive interference [4,43], we have not found a model that is easy to implement with existing hardware and software, and thus decided not to implement any such model in this work. The second limitation is naïve modeling of writes to main memory. Modern processors employ a write buffer that allows queueing writes and serving them only when there are no pending reads. Thus, without modeling write queues, the model becomes pessimistic. The third limitation is not modeling the prefetcher activity. As we show below, the second and third effects are marginal. By not modeling destructive interference we underestimate slowdown and hence our scheduler is less aggressive than it might have been with a more accurate model.

3.2. Evaluation of the model on a real system

We now evaluate the throughput model developed in Section 3.1 on a real system.

Methodology. We experiment with running multiple instances of every SPEC-CPU2006 benchmark in parallel. The experiments were performed on a quad-core and quad-thread Intel i5-2500, with 8 GB RAM in one module, 3.3 GHz internal core frequency with up to 3.7 GHz in Intel Turbo Boost mode, and a 1.33 MT/s memory bus. The server runs Linux kernel 3.2.0 with the *Completely Fair Scheduler* (CFS) [40]. Each data point was obtained by averaging at least five runs.

In each experiment, up to four instances of the same SPEC-CPU2006 benchmark are executed simultaneously, and we wait until all instances complete. Using identical instances minimizes the difference between the completion times of different threads, as done in [37].

Chip energy is measured using performance counters available in Sandy Bridge processors [16]. These counters estimate the energy consumed by the chip, including the cores, their private caches, the *Last Level Cache* (LLC), and the devices in the so-called uncore. Using the Running Average Power Limit interface [16], we read and reset the MSR_PKG_ENERGY_STATUS Machine Specific Register every 30 s, in order to prevent wraparound without excessive intervention to the benchmarks. These counters measure energy at a granularity of $\frac{1}{16}$ Joules. The energy measurements require very little CPU time, and hence, do not have a significant impact on our measurements.

We also measure the energy consumption of the entire system using an external power meter [46], connected to the computer's power supply. The external meter samples the energy consumed once per second, and we read it at the end of the benchmark. These measurements do not consume any CPU time.

Since an access to memory requires the transmission of one cache line, which in our platform consists of 64 bytes [17], and the quoted bandwidth is 1.33 million 64-bit transactions per second, we get that the bus bandwidth is:

$$\mu = \frac{1.33\ \mathrm{MT/s} \cdot 64}{64 \cdot 8} = 166{,}625{,}000 \left[\frac{\mathrm{accesses}}{\mathrm{s}}\right]. \quad (17)$$

Our model uses an estimate U of the memory bus utilization of all cores. We sample UNC_ARB_TRK_REQUEST.ALL to obtain the total number of accesses from the chip to main memory, $accesses^p$. Using this counter that can be sampled on any of the cores, we estimate the average bus utilization of all cores during a time interval Δt with the following expression:

$$U = \frac{1}{\mu \Delta t} accesses^p. \quad (18)$$

Our model also uses an estimate u_i of the memory bus utilization of each core i. Since memory bus utilization per core cannot be directly measured in our experimental platform, we estimate it by using other performance counters. We estimate the number of reads in core i, $reads_i^c$ by sampling OFFCORE_RESPONSE. ALL_READS. LLC_MISS. DRAM_N. Per-core counters for writes are not available in our platform, so we use chip-wide counters instead [16]. The number of writes from the processor chip to main memory, $writes^p$, is estimated by sampling UNC_ARB_TRK_REQUEST.EVICTIONS. Using these counters that are sampled on core i, we estimate the average bus utilization of core i during a time interval Δt by assuming that the writes are distributed in the same manner as the reads per core:

$$u_i = \frac{1}{\mu \Delta t}\left(reads_i^c + writes^p * \frac{reads_i^c}{accesses^p - writes^p}\right). \quad (19)$$

The effect of the assumption that writes are distributed similarly to reads on the total estimated bandwidth is low since the number of reads is usually much higher than the number of writes. Future processors should provide performance counters that allow more accurate bandwidth estimation per core.

Results. The solid red curve in Fig. 4 shows the throughput of running four identical threads in parallel as predicted by our model. As can be seen from the figure, when the solo bus utilizations are low, the throughput approaches 4 (full parallelism of the four cores), whereas the throughput approaches 1 (no effective parallelism) when the solo bus utilizations are high. In order to demonstrate the collision effect, we wrote a synthetic benchmark that utilizes a configurable portion of the memory bus when running solo on the processor. The benchmark accesses memory in a manner that always results in a cache miss, and performs an idle loop between accesses. We set the server frequency to its minimum, 1.6 GHz, to prevent the server's DVFS and Turbo Boost features from affecting the results. The throughput of running four identical synthetic benchmarks is shown in Fig. 4 (dashed green curve). The model predictions are within 10% of the measured results, and the standard deviation of the error is 3.3%. Since this benchmark has no destructive interference, the deviations can be attributed to the second and third limitations, namely memory controller, which attempts to join memory accesses, as well as to the prefetcher behavior.

The blue points in Fig. 4 represent the throughput of running four identical SPEC-CPU2006 benchmarks in parallel on our experimental platform. As can be seen, most points are well below the analytic model prediction, which is an upper bound. The differences can be attributed to destructive interference, where one thread evicts another's useful cache lines, causing the latter to require more bandwidth and more execution time.

4. Energy-friendly scheduler

We realize the concept of resource-aware scheduling in Linux kernel 3.2.0. We implement EFS as an extension of the default Linux scheduler, dubbed the *Completely Fair Scheduler* (CFS) [40]. Like virtually all OS schedulers, CFS strives to maximize CPU utilization, and runs a thread whenever one is available. As its name suggests,

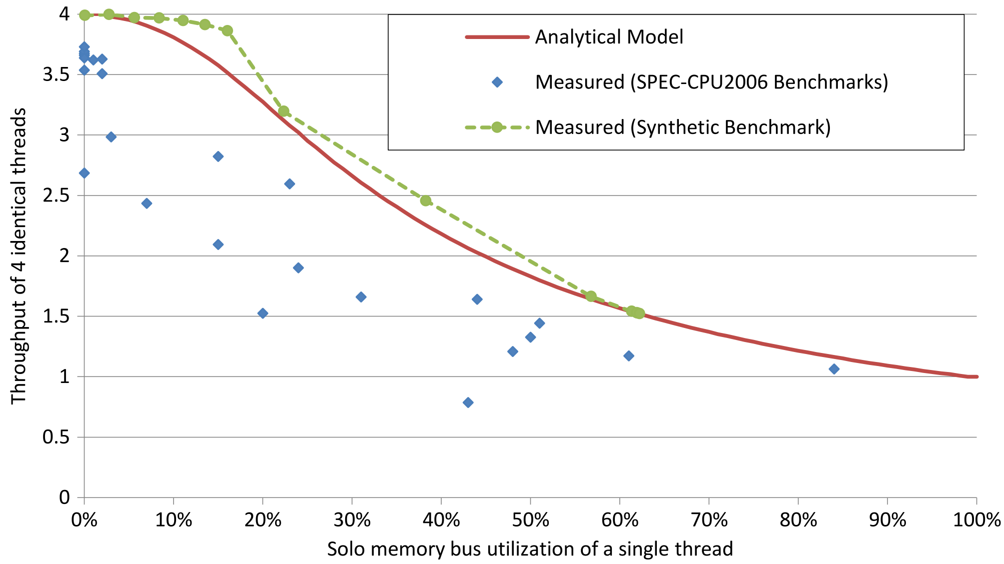


Fig. 4. Throughput of four identical threads running in parallel vs. their solo memory bus utilizations.

CFS provides fair allocation of processor time among running threads on each core. However, the scheduler runs separately and independently on each core. Hence, it does not attempt to ensure global fairness across cores [40]. A complementary load balancing mechanism [35] periodically attempts to balance the load among the cores.

The goals for our EFS are to decrease energy consumption without introducing starvation or harming responsiveness. We achieve the first goal by predicting future resource usage per thread as discussed in Section 4.1, and running only sets of threads whose combined requirements may be met by the platform, as described in Section 4.2. Starvation freedom and responsiveness are discussed in Section 4.3.

4.1. Resource usage prediction

We measure resource usage during scheduling events, which mark the end of a thread's *scheduling interval* (due to either preemption or the thread yielding the core). When thread i yields, we estimate its solo bandwidth b_i^k in interval k using Eq. (9), where U and u_i are measured using performance counters as explained in Section 3.2. We predict a thread's solo bandwidth requirement B_i^{k+1} for scheduling interval $k+1$ using the prediction B_i^k of the preceding scheduling interval k, the estimated solo bandwidth usage b_i^k in the preceding scheduling interval, the preceding interval's duration Δt_i^k, and a window size parameter t_0, as follows:

$$B_i^{k+1} = \begin{cases} \dfrac{b_i^k \Delta t_i^k + B_i^k (t_0 - \Delta t_i^k)}{t_0} & \Delta t_i^k < t_0 \\ b_i^k & \Delta t_i^k \geq t_0. \end{cases} \tag{20}$$

The window size t_0 determines the memory of the predictor. Low values of t_0 allow the predictor to quickly react to bandwidth usage changes, whereas higher values are more stable and smooth out short-lived changes. We chose $t_0 = 1$ ms in our experiments as it provided good results across many benchmarks. The initial prediction B_i^0 for a new thread is the inverse of the number of cores, i.e., 25% for a quad-core processor.

Note that u_i^k and B_i^k are sensitive to contention, as destructive interference leads to higher utilization values due to redundant cache misses. Such over-estimation offsets part of our modeling error, which does not take destructive interference into account, and causes our scheduler to also avoid bottlenecks that stem from destructive interference.

Given all solo bandwidths of the running threads, our scheduler calculates the throughput of the multicore by solving the set of equations given in Eq. (10). We solve the equations by using the *Multivariate Newton's Method* (MNM). Since the kernel does not support floating point operations, we implemented MNM using integer arithmetic. We perform at most 4 iterations, stopping earlier if the computation converges. The resulting utilizations $u_1 \dots u_n$ allow calculating the slowdowns given in Eq. (11) and the multicore throughput by using Eq. (12). The resulting throughput and solo bus utilizations are used for task selection as explained in Section 4.2.

When there are many threads, the cost of numerically solving Eqs. (10) may become significant. Lookup-tables can be used to reduce this cost. We did not find it necessary to implement such approximate methods in our four-core setup.

Since the scheduler code runs on each of the cores asynchronously, updates to shared variables are performed asynchronously as well. Due to the cost of synchronizing accesses to shared data, we decided to perform these accesses *unprotected*, that is, a thread may attempt to read a counter while another thread is updating it. While this can cause the scheduler to read incorrect values and incorrectly predict bandwidth usage, it does not impact the correctness of the scheduler, but rather only its efficiency.

4.2. Task selection

Like CFS, EFS operates independently on each core, which has its own set of runnable threads. On each core, CFS schedules threads according to their waiting times; threads with longer waiting times receive precedence over ones with shorter waiting times. Periodically or on-demand, the scheduler on a given core activates the load balancing routine.

EFS task selection is based on CFS with the following modification: when CFS would choose to schedule a thread, EFS first checks whether the system has enough resources to run the thread, given the thread's predicted resource usage and currently available system resources. This is done by solving Eqs. (10) and (16) as explained in Section 3.1.

If the resources suffice, the thread is executed as in CFS. In case the thread is deemed the longest starved thread in the system, it is also scheduled and other non-idle cores are notified, as discussed in the next section. If it is neither the longest starved nor has

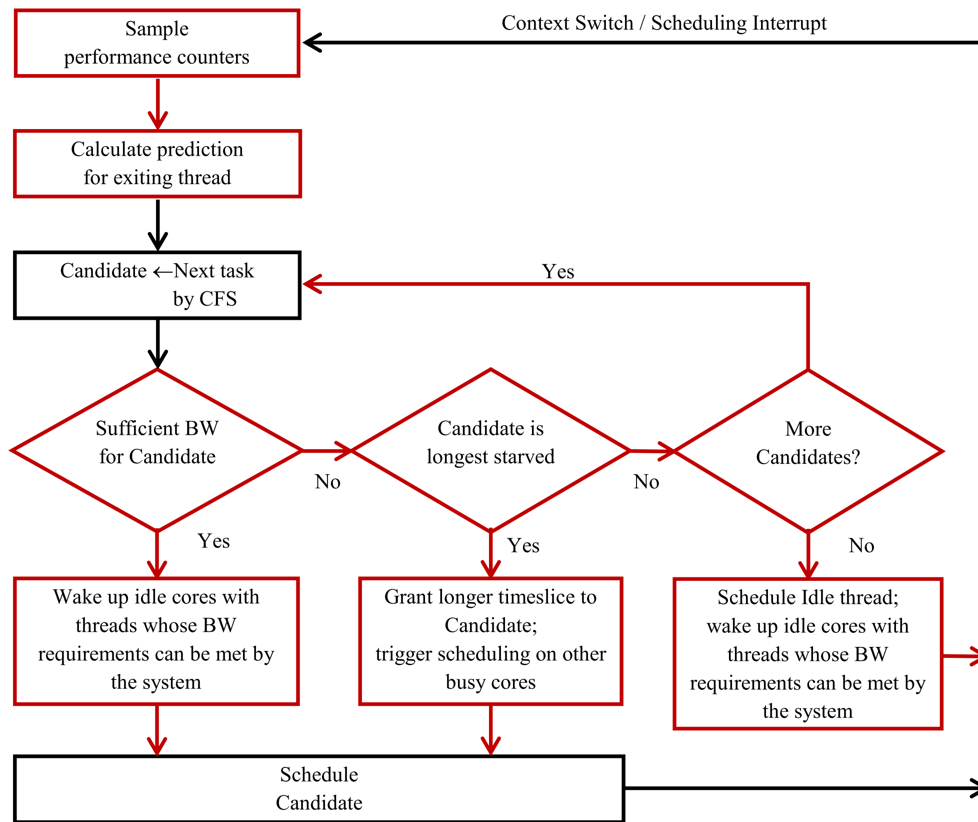


Fig. 5. Task selection in EFS on a single core. Black components are part of CFS, whereas red components are introduced by EFS. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

sufficient resources, it is skipped, and the next thread is evaluated. If no suitable thread is found, the core transitions to idle mode, where it typically enters an energy saving state. The task selection process of EFS on a single core is illustrated in Fig. 5.

When system resources are freed-up, it may become advantageous to run threads on other idle cores. Therefore, for each core EFS keeps track of the thread with the lowest resource requirements in O(1) time. When system resources become available, EFS triggers thread scheduling only on idle cores that have threads in their queues that can now run using the additional freed-up resources.

4.3. Responsiveness and starvation freedom

CFS strives to provide fairness in allotted CPU time among processes running on the same core. We argue that in a system where the chief bottleneck is not CPU time, fair allocation of other resources is no less important than fair allocation of CPU time. Moreover, there are scenarios where fair allocation of CPU time is inherently detrimental to performance and energy consumption. We have therefore chosen to forgo equal allocation of CPU time to threads on the same core. Instead, we opt to provide two important properties: (1) responsiveness and high priority for real-time threads, and (2) starvation-freedom.

Responsiveness and execution of real-time threads are not altered in any way by EFS, as they are not handled by CFS. Rather, they are scheduled separately by the Linux scheduler. Under the Linux scheduler, real-time threads have priority over all other threads, and they preempt any non-real-time thread when they need the CPU.

Starvation is more of a challenge. By enforcing an upper limit on overall resource utilization, EFS favors threads with low resource requirements. Threads with high resource requirements may wait for a long time, or even starve.

In order to detect and prevent starvation, we rely on data structures maintained by CFS. On each core, CFS keeps track of the time each runnable thread has run. Threads with higher priority have their tracked runtimes increase by a lesser extent than threads with low priority. CFS refers to these adjusted runtimes as *virtual runtimes*, or *vruntimes* for short. CFS achieves fairness in each core by scheduling the runnable thread that has the minimum *vruntime* on that core. This ensures that the virtual runtimes of threads on the same core are similar. Fairness among cores is not directly addressed by CFS, although it does employ a load balancing mechanism that attempts to equate the load across cores.

We consider a thread to be *starved* if:

(1) The thread's *vruntime* is minimal in its core; this means that CFS already identified this thread as the next candidate to run; and
(2) One of the following holds:
  (a) *Local starvation*—the difference between the thread's *vruntime* and the maximal *vruntime* among threads on the same core exceeds a certain threshold, t_{local}; or
  (b) *Global starvation*—the thread has not been scheduled to run for a time exceeding a second threshold, t_{global}.

The thresholds are calculated dynamically, as a function of CFS's scheduling period, which is in turn a function of the number of running threads on a core, n_c:

$$t_{period}(n_c) = \begin{cases} 18\text{ ms} & n_c < 5 \\ 4\text{ ms}\cdot n_c & n_c \geq 5. \end{cases} \tag{21}$$

The *local starvation* threshold used in (2a) is a constant multiplied by the scheduling period calculated according to the number of threads currently running on the core:

$$t_{local} = c_l t_{period}(n_c). \tag{22}$$

The *global starvation* threshold used in (2b) is a constant multiplied by the scheduling period calculated according to the number of threads currently running on the processor chip, n_p:

$$t_{global} = c_g t_{period}\left(n_p\right). \quad (23)$$

The global starvation threshold ensures that all threads in the system eventually run. The goal of the local starvation threshold is to mitigate unfairness among threads by reducing the lags in *vruntimes* on the same core. Unless stated otherwise, all benchmarks were run with $c_l = 20$ and $c_g = 5$.

We give threads that have been waiting for resources for a long time a chance to narrow the gaps from the remaining threads. We define the *longest starved* thread as the thread that has not run for the longest time among all of the *starved* threads. Once EFS encounters a longest starved thread, it schedules that thread to run immediately, as CFS would, regardless of the current resource usage. Note that this is a one-time event, and does not change the scheduler's operation in the future.

CFS distributes *timeslices* among threads according to their priorities, where the sum of *timeslices* of all threads is one scheduling period, $t_{period}(n_c)$. When a longest starved thread is scheduled, EFS compensates it with a longer *timeslice*, equal to the minimum between (a) one whole scheduling period: $t_{period}(n_c)$; and (b) the difference between the maximum *vruntime* in the core and the *vruntime* of the thread. After the starved thread runs for the increased time slice, the scheduler may preempt it, in case there is another longest starved thread or a thread with a lower value of *vruntime*.

To avoid forming a bottleneck when scheduling longest starved threads, EFS preempts all non-idle cores and requires them to re-check whether the system in its new state can allow the currently running threads to continue running.

Note that granting a larger *timeslice* to the longest starved thread causes other threads in the multicore to wait. Threads on the same core as the starved thread wait since they have already received more CPU time than the starved thread. Threads on other cores may also wait if the system does not have enough resources to run them. This is desirable when considering fairness at the resource level. The longest time a thread can wait because of this is bounded by the following expression, which captures the case when all threads in the system are starved:

$$t_{\max} = n^p \cdot t_{period}\left(n^p\right). \quad (24)$$

Note that interactive processes are unaffected by the starvation-freedom mechanism, as they spend most of the time sleeping. Thus, when an interactive thread wakes up, its *vruntime* has already accumulated a large lag from the other running threads, and it receives precedence. On the other hand, responsiveness of computational tasks may be degraded due to the additional waiting time. This degradation, however, is of less importance, since computational tasks are not sensitive to responsiveness, but to execution time and energy consumption, which are both improved by EFS, as we show below.

5. Experimental evaluation of EFS

In this section we use the experimental setup of Section 3 to compare our new scheduler to the baseline CFS. In Section 5.1 we tune our model parameters to our experimental setup. In Section 5.2 we study the effects of EFS on synthetic workloads, and in Section 5.3 we study the effects of EFS on SPEC-CPU2006 workloads.

5.1. Model fitting

In order to test the actual bus bandwidth limit, we wrote a micro-benchmark that incurs only cache misses. Our micro-benchmark achieves a maximum rate of 150 million accesses per second on our platform. The difference between the quoted bandwidth in Eq. (17) with one DDR3 channel and our measured result reflects certain inefficiencies of the memory controller, such as the overhead of transmitting bus requests, the overhead of opening and closing DRAM banks, DRAM refreshes, and the behavior of the prefetcher. We thus set $\mu = 150$ M.

We calibrate the power model from Section 3.1 by measuring the power consumed by running several identical instances of the same SPEC benchmarks on our system. The normalized standard deviation of the resulting power values is small: 3.3% for one active core, 2.8% for two active cores, 3.4% for three active cores, and 5.1% for four active cores. We thus use the average power values as the model parameters:

$$P_0 = 26.6\text{ W}; \quad P_1 = 22.6\text{ W}; \quad P_2 = 12\text{ W}; \quad P_3 = 12.3\text{ W}; \quad P_4 = 12.3\text{ W}. \quad (25)$$

5.2. Study of synthetic benchmarks

In the first experiment we evaluate our analytic model developed in Section 3.1. We compare it to the additive model proposed in prior research [1], where the joint bus utilization of two threads is estimated to be the sum of their solo utilizations; namely, the measured utilization of a thread under interference is assumed to be its solo utilization. Fig. 6 compares the estimation of the solo utilization of a thread computed by our scheduler to the one obtained using the additive model. In this experiment, we run two threads in parallel. The first runs the synthetic benchmark of Section 3.2 configured to utilize 57% of the bus when running solo (middle solid green line). The second is an interfering thread running the synthetic benchmark with a varying solo utilization (given in the x-axis). The solo bandwidth of the first thread estimated by our scheduler according to the model of Section 3.1 is depicted in the top dashed blue curve. The estimation of the additive model in this case is simply the measured utilization of the first thread, which is depicted in the bottom red curve in Fig. 6. We see that as interference increases, the additive model's estimation of the first thread's bus utilization strays significantly from the actual solo utilization (middle green line). In contrast, our model is able to estimate the solo utilization much more accurately for virtually any level of interference.

In the next experiment we run four instances of a synthetic benchmark configured to incur the maximum number of memory accesses, using CFS and EFS. For debugging purposes, we added the capability to log scheduling decisions and certain statistics. We use these to visualize the effects of EFS on core utilization. Fig. 7 shows a representative section of the experiment using CFS (on top) and EFS (on the bottom). We see that all cores are busy most of the time in CFS, whereas in EFS only one core is busy most of the time.

We now compare running the instances in parallel using CFS to running them on CFS serially, one after another, as well as to running them in parallel using EFS. The results are summarized in Table 1. Running four threads in parallel in EFS is more energy efficient than parallel execution on CFS by 47.6%. Runtime improves by 19.2% compared with parallel execution on CFS. Serial execution outperforms parallel execution on CFS due to destructive interference between the instances. We conclude that when the memory bus is fully utilized, adding more threads in parallel wastes energy without gaining any performance benefit.

5.3. Study of SPEC-CPU2006 benchmarks

Having shown that our scheduler mitigates contention effects in synthetic workloads designed to exhibit these effects, we turn

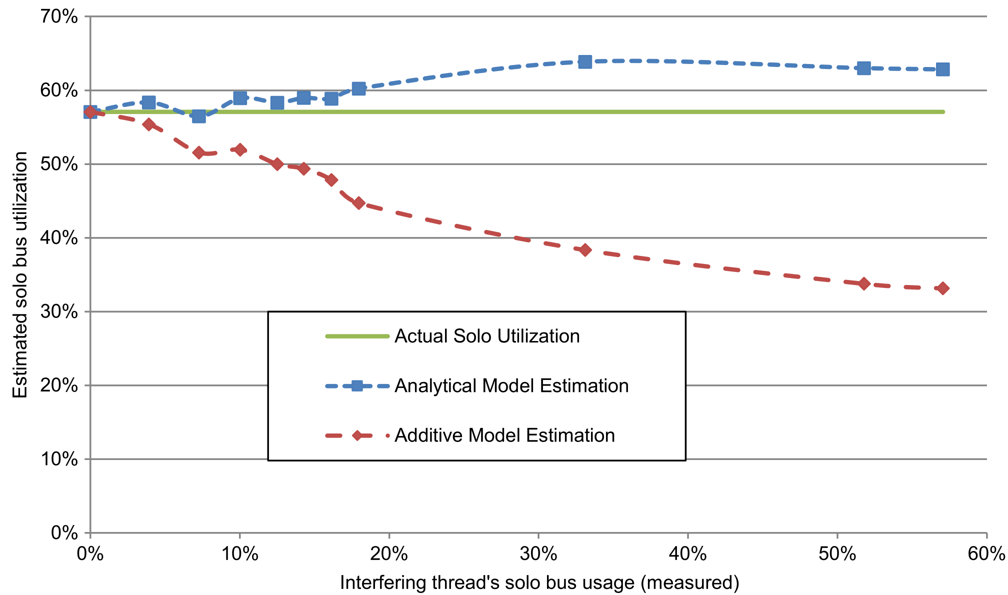


Fig. 6. Estimation of the solo utilization of a thread computed by our scheduler (top blue curve) compared to the one obtained using the additive model (bottom red curve), for a thread whose solo utilization is 57% (middle solid green line) that runs in parallel to an interfering thread with a variety of bus utilization levels (given in the *x*-axis). We see that our analytical model's estimations are very close to the actual solo utilization for virtually any level of interference. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

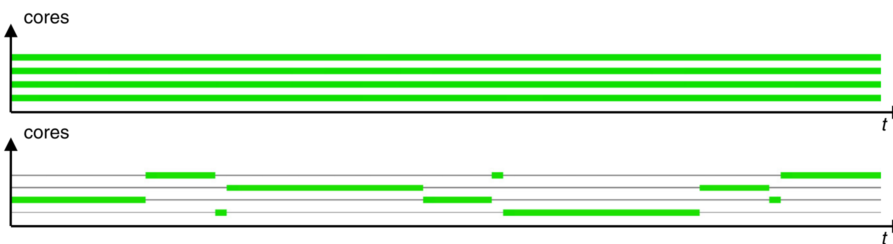


Fig. 7. Core utilization during 300 ms of running 4 instances of the synthetic benchmark on CFS (on top) and on EFS (on the bottom). We see that all four cores are busy all of the time on CFS, whereas EFS makes sure only one instance is active at any time due to the high resource requirements of each instance.

Table 1
Results for running four instances of the synthetic benchmark with EFS and CFS.

Benchmark and scheduler	Execution time (s)	CPU energy (J)	System energy (J)
Serial execution using CFS	21.84	527	904
Parallel execution using CFS	27.76	1376	2079
Parallel execution using EFS	23.48	527	1089
Improvement of parallel execution on EFS over CFS	19.2% speedup	61.7% energy reduction	47.6% energy reduction

to evaluate the scheduler with real workloads from the SPEC-CPU2006 benchmark suite.

We run four instances of each of the SPEC-CPU2006 benchmarks simultaneously. Fig. 8 shows the energy savings and speedup achieved by EFS compared to CFS in each of the benchmarks. We see that the energy consumption of the system (depicted by the leftmost, red bars), which is our target metric, improves in 13 out of 25 benchmarks. On average, system energy is reduced by 7.7%. Although not targeted by our scheduler, average CPU energy (represented by the second, blue bars) is reduced by 12.3% and the average system energy-delay (not shown) improves by 5.4%. The average slowdown is 1.9% (represented by the third, green bars).

Most benchmarks (13 out of 25) are practically unaffected by EFS and consume energy within 3% of CFS. The little extra energy consumed by the group of unaffected benchmarks in Fig. 8 is attributed to the added complexity of our new scheduler. The affected benchmarks (12 out of 25) exhibit a 17.6% average reduction in system energy with a 2.1% average slowdown. We conclude that while having relatively little impact in benchmarks which do not stress the memory bus, our scheduler is very effective at detecting and preventing excessive resource demands.

The greatest system energy savings of 32.2% were achieved by 462.libquantum, which also exhibited a slowdown of 3.7%. The greatest speedup of 4.2% was achieved by 459.GemsFDTD, while system energy was reduced by 24.8%. In many cases, EFS improves both energy and throughput.

It is interesting to compare the improvements offered by our scheduler over those that could have been achieved by statically activating only a subset of the cores on CFS. Fig. 9 shows a comparison of energy savings and speedup when activating one, two and three out of four cores and running four instances of the same SPEC-CPU2006 benchmarks on CFS. We see that EFS outperforms all of the static methods.

We next evaluate EFS with workloads comprising of a mix of different programs. The results in Fig. 10 show that EFS is also effective for mixed workloads with high resource requirements, achieving on average 23.7% system energy savings and 0.3%

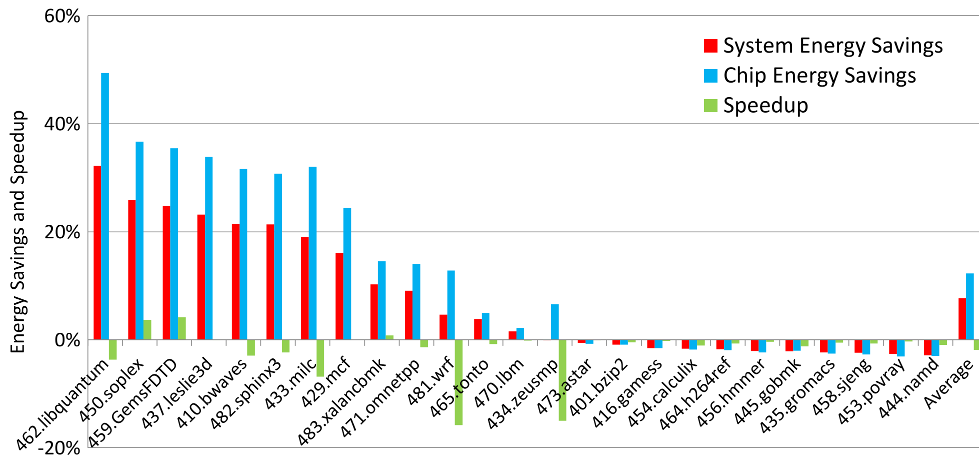


Fig. 8. System energy savings, chip energy savings and speedup for four instances of each SPEC-CPU2006 benchmark. The values represent the improvement of EFS over CFS. Negative values imply degradation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

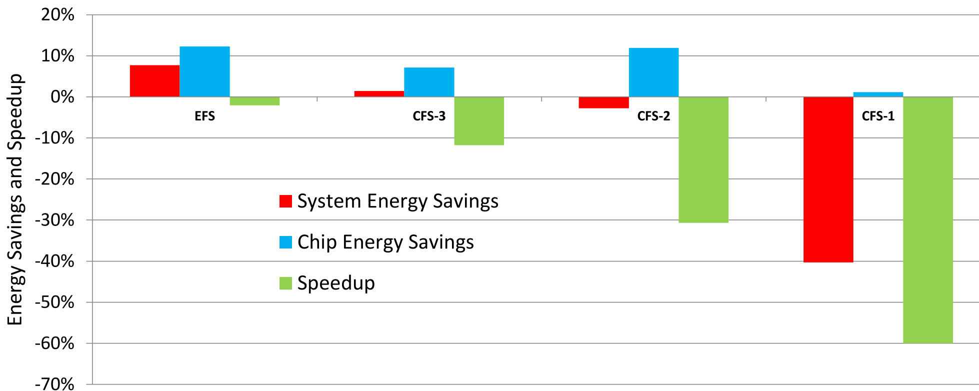


Fig. 9. Average energy savings and speedup of EFS and CFS where only a subset of the cores are active (CFS-x depicts CFS with x active cores), over CFS with all four cores active, running four instances of each SPEC-CPU2006 benchmark. Negative values imply degradation.

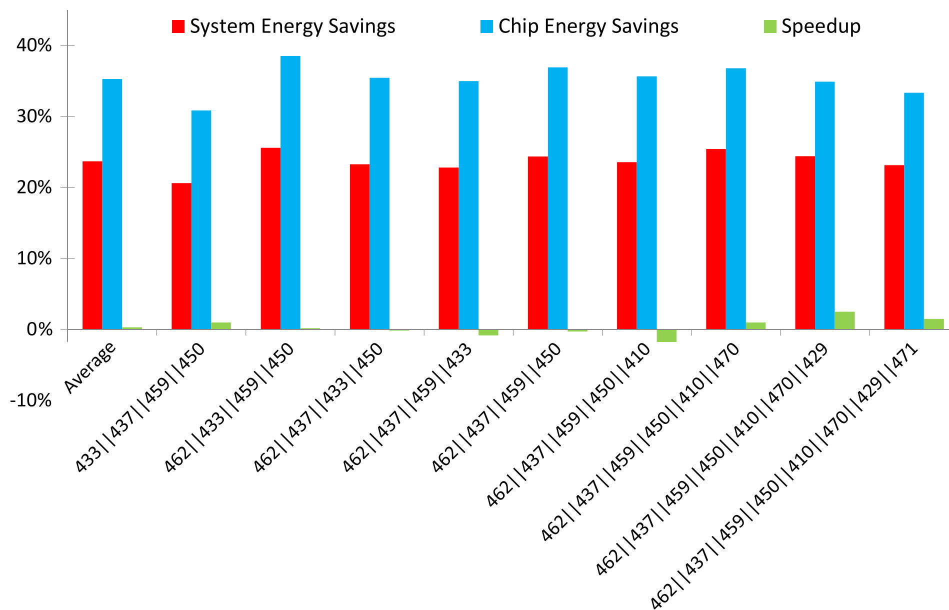


Fig. 10. Energy savings and speedup of EFS over CFS for workloads with mixed SPEC-CPU2006 benchmarks. The values represent the improvement of EFS over CFS. Negative values imply degradation.

speedup. The effectiveness of EFS is consistent even when the number of programs exceeds the number of cores (four rightmost columns). EFS has no effect in other experiments (not shown) with mixes that do not stress the memory bus.

6. Conclusions and future work

We have shown that interference among threads can result in both increased energy consumption and performance degradation.

In many cases, it can be more energy efficient to run applications serially than to run them in parallel in a multicore system. This is because when several threads run in parallel, bottlenecks often begin to form on critical resources. We identified the memory bus utilization as a critical bottleneck resource. We have demonstrated an operating system scheduler that prevents bottlenecks in this resource. Our scheduler achieved up to 32.2% system energy reduction for multiple copies of the SPEC-CPU2006 benchmarks running in parallel, compared to the standard Linux scheduler. These results were achieved on a real multicore system with four cores, with our new variant of the Linux scheduler. The energy was measured using an external power meter. It is expected that energy savings will be even higher with more than four cores, since our scheduler will keep more cores idle compared to the standard Linux scheduler. The concepts we presented in this paper can be used in today's commercial and open source scheduler implementations.

This work can be extended to further improve energy efficiency. First, CPU providers should expose accurate per-core performance counters such as per-core memory bus utilization and per-core energy consumption. CPU providers should also expose counters that aid in predicting destructive interference among threads, as well as models for the memory controller. The more counters available, the better the operating system can optimize the system for energy efficiency. Second, it may be valuable to consider bottlenecks on other shared resources such as the memory capacity, network, disk usage, all levels of cache, shared accelerators on the chip, GPU, etc.

We considered in this research workloads consisting of independent tasks. When considering multithreaded programs, delaying critical threads due to insufficient resources can have an adverse effect on throughput and system energy. An interesting direction for future work can be to focus on the prevention of bottlenecks with multithreaded workloads.

Another important direction for further research is attempting to determine whether resource management concepts such as those presented in this paper should reside in hardware, in software, or in both. A hardware resource manager could potentially realize more gains due to the ability to react quickly to the dynamically changing resource demands of applications.

Acknowledgments

This research was supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Hasso-Plattner Institute (HPI). We thank Ahmad Yasin from Intel Corporation for his insightful comments.

References

[1] C.D. Antonopoulos, D.S. Nikolopoulos, T.S. Papatheodorou, Realistic workload scheduling policies for taming the memory bandwidth bottleneck of SMPs, in: High Performance Computing-HiPC 2004, Springer, Berlin, Heidelberg, 2005, pp. 286–296.

[2] R. Azimi, D.K. Tam, L. Soares, M. Stumm, Enhancing operating system support for multicore processors by using hardware performance monitoring, SIGOPS Oper. Syst. Rev. (2009).

[3] M. Bhadauria, S.A. McKee, An approach to resource-aware coscheduling for CMPs, in: Proceedings of the International Conference on Supercomputing, ICS, 2010, pp. 189–199.

[4] D. Chandra, F. Guo, S. Kim, Y. Solihin, Predicting inter-thread cache contention on a chip multi-processor architecture, in: Proceedings of the Symposium on High Performance Computer Architecture, HPCA, 2005, pp. 340–351.

[5] S. Chen, P.B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry, C. Wilkerson, Scheduling threads for constructive cache sharing on CMPs, in: Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA, 2007, pp. 105–115.

[6] J.A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic', J. Kubiatowicz, Resource management in the tessellation manycore OS, in: Proc. of the 2nd USENIX Workshop on Hot Topics in Parallelism, HotPar'10, Berkeley, CA, USA, 2010.

[7] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware scheduling for heterogeneous datacenters, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13, ACM, New York, NY, USA, 2013, pp. 77–88.

[8] E. Ebrahimi, C.J. Lee, O. Mutlu, Y.N. Patt, Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems, in: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, ACM, New York, NY, USA, 2010.

[9] S. Eyerman, L. Eeckhout, Probabilistic job symbiosis modeling for SMT processor scheduling, ACM SIGPLAN Not. 45 (2010) 91–102.

[10] P.B. Galvin, G. Gagne, A. Silberschatz, Operating System Concepts, John Wiley & Sons, Inc., 2013.

[11] F. Guo, Y. Solihin, L. Zhao, R. Iyer, Quality of service shared cache management in chip multiprocessor architecture, ACM Trans. Archit. Code Optim. 7 (3) (2010) Article 14, Pub. date.

[12] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, U.C. Weiser, Many-core vs. many-thread machines: Stay away from the valley, Comput. Archit. Lett. 8 (1) (2009) 25–28.

[13] W. Heirman, T.E. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, L. Eeckhout, Undersubscribed threading on clustered cache architectures, in: Proceedings of the International Symposium on High-Performance Computer Architecture, HPCA 2014.

[14] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, J. Moses, Rate-based QoS techniques for cache/memory in CMP platforms, in: Proceedings of the International Conference on Supercomputing, ICS, 2009, pp. 479–488.

[15] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, D. Newell, PIRATE: QoS and performance management in CMP architectures, SIGMETRICS Perform. Eval. Rev. 37 (2010) 3–10.

[16] Intel 64 and IA-32 Architecture Software Developer's Manual.

[17] Intel Core i5-2500 datasheet: http://ark.intel.com/products/52209/Intel-Core-i5-2500-Processor-6M-Cache-up-to-3_70-GHz.

[18] Intel datasheets in http://ark.intel.com.

[19] ITRS 2012 Assembly and Packaging Tables and Lithography Tables.

[20] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, S. Reinhardt, QoS policies and architecture for cache/memory in CMP platforms, SIGMETRICS Perform. Eval. Rev. 35 (2007) 25–36.

[21] Y. Jiang, X. Shen, J. Chen, R. Tripathi, Analysis and approximation of optimal co-scheduling on chip multiprocessors, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT, 2008, pp. 220–229.

[22] Y. Jiang, K. Tian, X. Shen, Combining locality analysis with online proactive job co-scheduling in chip multiprocessors, in: Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC, 2010, pp. 201–215.

[23] H. Jonkers, Queueing models of parallel applications: the Glamis methodology, in: Computer Performance Evaluation Modelling Techniques and Tools, Springer, Berlin, Heidelberg, 1994, pp. 123–138.

[24] M. Kambadur, T. Moseley, R. Hank, M.A. Kim, Measuring interference between live datacenter applications, in: proceedings of SC12, Salt Lake City, Utah, USA, November 10–16, 2012.

[25] S. Kim, D. Chandra, Y. Solihin, Fair cache sharing and partitioning in a chip multiprocessor architecture, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT, 2004, pp. 111–122.

[26] S.G. Kim, H. Eom, H.Y. Yeom, Virtual machine consolidation based on interference modeling, J. Supercomput. 66 (3) (2013) 1489–1506.

[27] Y. Kim, M. Papamichael, O. Mutlu, M. Harchol-Balter, Thread cluster memory scheduling: Exploiting differences in memory access behavior, in: Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2010, pp. 65–76.

[28] L. Kleinrock, Computer Applications, Vol. 2, Queueing Systems, Wiley, 1976.

[29] E. Koukis, N. Koziris, Memory bandwidth aware scheduling for SMP cluster nodes, in: PDP'05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 187–196.

[30] J. Li, J.F. Martínez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, in: Proceedings of the International Symposium on High-Performance Computer Architecture, Austin, TX, HPCA 2006.

[31] J. Mars, L. Tang, R. Hundt, Whare-map: Heterogeneity in homogeneous warehouse-scale computers, in: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA 2013.

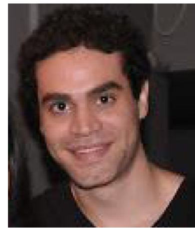
[32] J. Mars, L. Tang, R. Hundt, K. Skadron, M.L. Soffa, Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011, pp. 248–259.

[33] J. Mars, L. Tang, M.L. Soffa, Directly characterizing cross core interference through contention synthesis, in: Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC, 2011, pp. 167–176.

[34] J. Mars, N. Vachharajani, R. Hundt, M.L. Soffa, Contention aware execution: online contention detection and response, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO, 2010, pp. 257–265.

[35] W. Mauerer, Professional Linux Kernel Architecture, Wiley India Pvt. Limited, 2008.
[36] A. Merkel, J. Stoess, F. Bellosa, Resource-conscious scheduling for energy efficiency on multicore processors, in: Proceedings of the 5th European Conference on Computer Systems, EuroSys'10, ACM, New York, NY, USA, 2010, pp. 153–166.
[37] T.Y. Morad, A. Kolodny, U.C. Weiser, Task scheduling based on thread essence and resource limitations, J. Comput. 7 (1) (2012).
[38] M. Moreto, F.J. Cazorla, A. Ramirez, R. Sakellariou, M. Valero, FlexDCP: a QoS framework for CMP architectures, ACM SIGOPS Oper. Syst. Rev. 43 (2009) 86–96.
[39] O. Mutlu, T. Moscibroda, Stall-time fair memory access scheduling for chip multiprocessors, in: Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2007, pp. 146–160.
[40] C.S. Pabla, Completely fair scheduler, Linux J. (184) (2009).
[41] K.K. Pusukuri, D. Vengerov, A. Fedorova, V. Kalogeraki, Fact: a framework for adaptive contention-aware thread migrations, in: Proceedings of the International Conference on Computing Frontiers, CF, 2011.
[42] S. Reda, R. Cochran, A.K. Coskun, Adaptive power capping for servers with multithreaded workloads, IEEE Micro 32 (5) (2012) 64–75.
[43] A. Sandberg, D. Black-Schaffer, E. Hagersten, A simple statistical cache sharing model for multicores, in: Proceedings of the 4th Swedish Workshop on Multi-Core Computing, 2011, pp. 31–36.
[44] D.J. Sorin, V.S. Pai, S.V. Adve, M.K. Vernon, D.A. Wood, Analytic evaluation of shared-memory systems with ILP processors, ACM SIGARCH Comput. Archit. News 26 (3) (1998) 380–391. IEEE Computer Society.
[45] A. Vega, A. Buyuktosunoglu, P. Bose, SMT-centric power-aware thread placement in chip multiprocessors, in: 2013 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2013, pp. 167–176.
[46] Wattsup meters. http://www.wattsupmeters.com.
[47] X. Zhang, S. Dwarkadas, K. Shen, Hardware execution throttling for multi-core resource management, in: Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX'09. USENIX Association, Berkeley, CA, USA, 2009, pp. 23–23.
[48] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, CPI^2: CPU performance isolation for shared compute clusters, in: SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic, 2013, pp. 379–391.
[49] S. Zhuravlev, S. Blagodurov, A. Fedorova, Addressing shared resource contention in multicore processors via scheduling, in: ASPLOS'10: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, pp. 129–142.
[50] S. Zhuravlev, J.C. Saez, S. Blagodurov, A. Fedorova, M. Prieto, Survey of scheduling techniques for addressing shared resources in multicore processors, ACM Comput. Surv. 45 (1) (2012).

Tomer Y. Morad is a postdoc at the Runway program of the Jacobs Technion-Cornell Institute of Cornell Tech. Tomer has co-founded DatArcs, a provider of dynamic tuning technology for boosting server performance and energy efficiency. Prior to serving as CEO of DatArcs, Tomer co-founded transSpot, a provider of systems for digital signage and public transportation. Tomer held senior positions in the industry, including CEO of transSpot, Chief Security Officer at Horizon Semiconductors, and a technical team leader at an intelligence unit in the IDF. Tomer earned his B.Sc., M.Sc., and Ph.D. in Electrical Engineering from the Technion — Israel Institute of Technology in 2001, 2005 and 2015 respectively. His main research interests are energy-efficient resource allocation in server systems.



Noam Shalev is a Ph.D. Candidate at the Technion - Israel Institute of Technology, advised by Prof. Idit Keidar. Noam earned his B.Sc. in Electrical Engineering from the Technion in 2012 and graduated summa cum laude. His record includes several paper publications and joint works with leading research groups in Microsoft Research and IBM Research. Noam also co-founded his own start-up and took a major role in the OneDay Social Volunteering initiative. Funded by the Hasso-Plattner Institute, his research spans operating systems, machine learning, computer security and fault tolerance.

Idit Keidar is a Professor and Associate Dean at the Viterbi Faculty of Electrical Engineering at the Technion, where she heads the Networked Software Systems Laboratory (NSSL). She received her B.Sc. (summa cum laude), M.Sc. (summa cum laude) and Ph.D. at the Hebrew University of Jerusalem in 1992, 1994, and 1998 resp. She was a postdoctoral research associate at MIT's laboratory for Computer Science, where she held post-doctoral fellowships from Rothschild Yad-Hanadiv and NSF CISE. Prof. Keidar was awarded the Yanai Award for Excellence in Academic Education, the Muriel and David Jackow Award for Excellence in Teaching, the David Dudi Ben-Aharon Research Award, the Allon Fellowship, the Rothschild Yad-Hanadiv fellowship for postdoctoral studies, and a Wolf Foundation Prize for Ph.D. students. Prof. Keidar's research is broadly in distributed and concurrent algorithms and systems, as well as fault-tolerant network-based computing. She has served on over 30 program committees, including as PC Chair of DISC 2009 and SYSTOR 2015, as a columnist for SIGACT News, and a guest editor for Distributed Computing, and is currently serving on the editorial board of IEEE CAL.

Avinoam Kolodny is a 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 professor at the Viterbi Faculty of Electrical Engineering of the Technion IIT. 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 to 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 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.