Survey of Energy-Cognizant Scheduling Techniques

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto

Abstract—Execution time is no longer the only metric by which computational systems are judged. In fact, explicitly sacrificing raw performance in exchange for energy savings is becoming a common trend in environments ranging from large server farms attempting to minimize cooling costs to mobile devices trying to prolong battery life. Hardware designers, well aware of these trends, include capabilities like DVFS (to throttle core frequency) into almost all modern systems. However, hardware capabilities on their own are insufficient and must be paired with other logic to decide if, when, and by how much to apply energy-minimizing techniques while still meeting performance goals. One obvious choice is to place this logic into the OS scheduler. This choice is particularly attractive due to the relative simplicity, low cost, and low risk associated with modifying only the scheduler part of the OS. Herein we survey the vast field of research on energy-cognizant schedulers. We discuss scheduling techniques to perform energy-efficient computation. We further explore how the energy-cognizant scheduler's role has been extended beyond simple energy minimization to also include related issues like the avoidance of negative thermal effects as well as addressing asymmetric multicore architectures.

Index Terms—Survey, shared resource contention, thread level scheduling, power-aware scheduling, thermal effects, asymmetric multicore processors, cooperative resource sharing

1 Introduction

Increasingly widespread use of computer systems and rising electricity prices has brought energy efficiency to the forefront of the current research agenda. Energy consumption has become a key metric for evaluating how good a computer system is, alongside more traditional performance metrics like the speed of execution. A lot of effort is focused on building more energy-efficient devices, but even as the hardware becomes more energy efficient, the onus of extracting the maximum efficiency out of it very often falls on the software. This is particularly true about the CPU devices, which consume nearly half of energy in server systems. Modern processors provide "knobs" allowing to trade execution speed for power efficiency, but the decision of how to use these knobs to minimize the impact on speed while saving as much energy as possible falls onto the runtime resource manager, such as the operating system. Tuning the settings of energy-management knobs often involves deciding when and where to run each software thread and at what speed—these decisions are natural to make as part of conventional process scheduling routines. Therefore, we refer to CPU energy-management policies and algorithms employed in the software runtime as *energy-cognizant scheduling*. Energy-cognizant scheduling is the subject of this survey, and we focus specifically of managing energy consumption of the CPU.

We examine three types of hardware mechanisms allowing us to manage CPU energy consumption: 1) Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM), 2) thermal management, and 3) asymmetric multicore designs.

DVFS and DPM allow us to dynamically throttle the voltage and frequency of the CPU or to temporarily suspend its operation and put it in a low-power state. As we throttle or suspend the processor we inevitably sacrifice execution speed, and so the key challenge in designing DVFS/DPM control algorithms is to find the setting that saves the most power while sacrificing the least speed. This requires understanding applications' characteristics and the intricacies of their interaction with the hardware, and that is why DVFS/DPM control algorithms are nontrivial to design and are undeniably interesting.

We then survey runtime algorithms that perform thermal management. Although in a somewhat indirect fashion, processor temperature plays a crucial role in energy consumption, because as the temperature becomes high, the system needs to run fans to cool off the processor. Cooling consumes electricity, and so managing system temperature can reduce the amount of energy spent on cooling. Thermal management relies primarily on altering the physical placement of threads on cores so as to avoid thermal hotspots and temperature gradients (i.e., when one area on the chip is much hotter than another). Thermal management algorithms rely on curious and unexpected effects on processor temperature created by the instruction

- *S. Zhuravlev and A. Fedorova are with the School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, BC V5A 1S6, Canada. E-mail: sza21@sfu.ca, fedorova@cs.sfu.ca.*
- *J.C. Saez and M. Prieto are with the Facultad de Ciencias Fisicas, Complutense University of Madrid, Avda. Complutense s/n, Ciudad Universitaria, Madrid 28040, Spain. E-mail: jcsaezal@fdi.ucm.es, mpmatias@dacya.ucm.es.*
- *S. Blagodurov is with the Simon Fraser University, 3795 Pandora Street, Burnaby, BC, Canada V5C 2A4. E-mail: sergey_blagodurov@sfu.ca.*

Manuscript received 7 Mar. 2011; revised 19 Nov. 2011; accepted 27 Nov. 2011; published online 5 Jan. 2012.
Recommended for acceptance by A. Zomaya.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-03-0128.
Digital Object Identifier no. 10.1109/TPDS.2012.20.

mix, the physical placement of threads on the chip and threads' relative proximity to one another.

While DVFS and DPM allow dynamically throttling the speed and thus the power consumption of the processor, even greater energy efficiency can be achieved if CPU cores are engineered from the start to be low power. However, low-power cores typically offer smaller and weaker execution engines than their high-power counterparts, making significant sacrifices in processing speed. An interesting attempt at resolving this dilemma is offered by *asymmetric* systems. These systems are built with both low-power and high-power cores on the same chip. Both core types are able to execute the same binary, but they differ in microarchitecture, power profiles, and performance.[1] As a result, the thread scheduler can use this asymmetry as a knob for controlling speed-energy tradeoff. By scheduling threads that do not benefit from advanced features of high-power cores on low-power cores and vice versa, the scheduler can maximize performance within the system's frugal power budget.

While the algorithms discussed in this paper were developed for different types of energy-management mechanisms their unifying theme is that they work by dynamically monitoring properties of the workload, making decisions that consider how the characteristics of the workload interplay with those of the hardware, and controlling the configuration and allocation of CPU cores so as to make the best tradeoff between performance and power consumption.

This survey is organized as follows: Section 2 introduces basic concepts on power and energy consumption and presents a classification of the different techniques analyzed in this survey. Section 3 covers scheduling algorithms that use DVFS and DPM. Section 4 discusses thermal management. Section 5 focuses on scheduling for asymmetric machines. We conclude and discuss what we see as the future directions of energy-cognizant scheduling research in Section 6.

2 Background

2.1 Power Dissipation Basics

CMOS devices are the building blocks of most general-purpose computing systems today. Power consumed in CMOS circuits can be divided into three components: *dynamic*, *static*, and *short-circuit* power. For many years *dynamic power dissipation*, which occurs due to the switching of transistors, has been the dominant factor in CMOS power consumption. Dynamic power can be approximated with the following formula:

$$P_d = C_l \cdot N_{sw} \cdot V_{dd}^2 \cdot f, \tag{1}$$

where C_l is the load capacitance, a significant percentage of which is wire related, N_{sw} is the average number of circuit switches per clock cycle, V_{dd} is the supply voltage, and f is the clock frequency.

The shrinking of the feature size in the latest technology generations has enabled to substantially reduce the supply voltage (V_{dd}), which reduces power dissipation quadratically. Unfortunately, decreasing the supply voltage increases the circuit delay and so the circuits cannot be driven at the same clock frequency. More specifically, the clock frequency is almost linearly related to the supply voltage as follows:

$$f = k \cdot \frac{(V_{dd} - V_{th})^2}{V_{dd}}, \tag{2}$$

where k is a constant and Vth is the threshold voltage [1], [2]. In turn, P_d is roughly cubically related to f: $P_d \approx C_l \cdot N_{sw} \cdot \frac{f^3}{k^2}$. As a result, a reduction in the supply voltage and the clock frequency reduces dynamic power cubically (and dynamic energy quadratically) at the expense of up to a linear increase of execution time. For example, consider an application that takes 10 seconds to complete at a given processor frequency f. At this frequency the dynamic power and energy of the task is denoted by P_d and E_d, respectively, where $E_d = P_d \cdot \text{completion\_time}$ (hence, $E_d = P_d \cdot 10$). Suppose further that, when reducing the processor frequency and supply voltage by half, this task takes 20 seconds to complete. The new dynamic power P'_d and energy E'_d consumed by the task in this scenario would be as follows:

$$\begin{aligned} P'_d &= C_l \cdot N_{sw} \cdot \left(\frac{V_{dd}}{2}\right)^2 \cdot \frac{f}{2} \\ &= \frac{1}{8} \cdot C_l \cdot N_{sw} \cdot V_{dd}^2 \cdot f = \frac{1}{8} \cdot P_d, \end{aligned}$$

$$E'_d = P'_d \cdot 20 = \frac{5}{2} \cdot P_d = \frac{5}{2} \cdot \frac{E_d}{10} = \frac{1}{4} \cdot E_d.$$

The second component of power dissipation, *static (or leakage) power*, has to do with the existence of leakage current flowing between the power source and the ground. Subthreshold leakage, the main subcomponent of leakage power, represents the power dissipated by a transistor whose gate is intended to be off. In the latest process generations subthreshold leakage power has increased exponentially with respect to previous generations, mainly due to joint reductions in the supply and the threshold voltage.[2] Currently, leakage power constitutes 20-40 percent of the total power dissipation [3]. Of special attention is the exponential dependence of leakage power on temperature. As the system's temperature rises, leakage power becomes a more significant portion of the total power. Most research efforts to reduce leakage power consumption are taking place at the process level [4], [5] and at the microarchitectural level [6], [7], [8], and so they are not within the scope of our survey. Nevertheless due to the relation between temperature and static power, techniques described in Section 4, which strive to control the processor temperature, contribute to reducing static power as well.

Finally, short circuit or "glitching" power is the power dissipated when both the n and p transistors of a CMOS gate are conducting simultaneously. Because short-circuit power is comparatively insignificant to dynamic and static power, this component of power consumption has not drawn as much attention as the others from the research community and the industry.

1. Heterogeneous systems where cores of different types support different instruction sets are not discussed in this survey.

2. It is evident from (1) and (2) that V_{dd} and Vth have to be scaled down simultaneously in order to maintain performance while saving power.

TABLE 1
Summary of Runtime Power Management Techniques

Technique	Goals	Key Ideas	Approaches
DVFS/DPM	Reduce operating voltage/frequency or power state while minimizing or bounding effect on performance.	Some applications suffer less performance loss from frequency reduction than others. Real-time and deadline-driven applications have performance slack: no benefit from performance gains past meeting their deadline.	Processor frequency is selected based on: 1) a performance model for relationships between performance and power based on application characteristics; 2) a static processor-specific model
Thermal management	Reduce thermal emergencies and thermal gradients by proactively or reactively moving "guilty" threads away from areas that are likely to overheat.	The type of computation that threads execute affects how much heat they are going to generate.	1) Reactively move threads from hot to cool areas; 2) pro-actively avoid clustering those threads that are more likely to produce heat.
Asymmetry-Aware Scheduling	Assign threads to cores so performance is maximized	The relative benefit of running on a "fast" vs. "slow" core depends on the type of code being executed: e.g., compute-intensive vs. memory-intensive, parallel vs. serial.	1) Trial-and-error: schedule threads, measure performance, adjust; 2) Analytical modeling: predict performance of a thread on a core type given threads characteristics measured online.

2.2 Energy Management at Runtime

Examining the relationships in (1) justifies three main research directions aiming to reduce system energy consumption via software techniques used at runtime. The first group of solutions is based on **Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM)**. These solutions dynamically vary the processor voltage and frequency. Since reducing voltage and frequency may lower performance, the DVFS/DPM-based solutions consider relationship between frequency and performance, taking into account the fact that some applications slow down less than others when processor frequency is reduced.

The second group of solutions, **Thermal Management**, has to do with the fact that processor temperature has a significant impact on energy consumption. For example, when a processor heats up, the system must increase activity of the cooling infrastructure (e.g., fans), which consume energy. This group of solutions relies on the observation that different applications have different effect on processor temperature and leverages interesting spatial dependencies when it comes to temperature control. For instance, running two heat-generating threads on adjacent cores will create a greater thermal gradient than running these threads on separate parts of the chip. Thermal management algorithms thus perform spatial and temporal placement of threads so as to minimize thermal emergencies and to reduce thermal gradients, which occur when the variation in temperature on different parts of the chip is high.

The third group of solutions, **Asymmetry-Aware Scheduling**, has to do with systems that are *asymmetric* in nature. Fundamentally, the system's power consumption depends on its microarchitecture: for instance, systems clocked at higher frequencies and featuring more complex pipelines and larger caches consume more power than smaller processors with simpler features. However, simpler and smaller processors also offer lower performance. To resolve the energy-performance tradeoff, some researchers proposed to build systems that include both complex, fast, and power-hungry cores as well as simple, lean and power-efficient cores. These systems are called asymmetric. In this survey, we for the most part discuss symmetric-ISA (Instruction Set Architecture) asymmetric-performance systems—where all cores can run the same application binary, but offer different performance.

In symmetric-ISA asymmetric-performance systems the onus of deciding which type of core should be used to run a particular thread falls on the operating system scheduler. Intelligent scheduling algorithms that optimize system energy and performance are thus crucial.

Another category of asymmetric systems is where asymmetry is not an explicit design feature, but occurs because of routine hardware faults (e.g., cores that were meant to run at identical frequencies actually run at different ones). These systems share similar challenges as explicitly asymmetric systems. However, because of the dynamic nature of their asymmetry (e.g., hardware faults making the system asymmetric can occur anytime during the execution) we logically place these solutions in the same category as DPM and DVFS.

To summarize, energy-management optimizations in the thread scheduler can be classified in three categories according to the mechanism upon which they rely: 1) Dynamic Power Management and Dynamic Voltage/Frequency Scaling, 2) Thermal Management, and 3) Asymmetry-Aware Scheduling. Table 1 summarizes the main challenges addressed by the these solutions as well as key ideas behind them. Detailed discussion of these techniques is presented in the rest of the paper, in Sections 3, 4, and 5, respectively.

3 Dynamic Voltage/Frequency Scaling and Dynamic Power Management

3.1 Basics of DPM and DVFS

Dynamic Power Management (DPM) exploits the fact that certain electronic components/devices may remain idle a

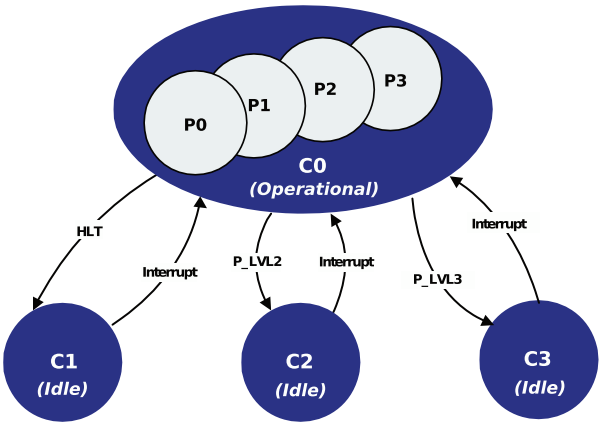

Fig. 1 A processor can be either in an inactive very-low-power $(C1, C2, \ldots, Ck)$ state, or in an active state (C0) where it can execute instructions. In turn, for processors in the C0 state a performance/energy tradeoff can be made via multiple performance (Px) states, typically characterized by different clock frequencies and supply voltages.

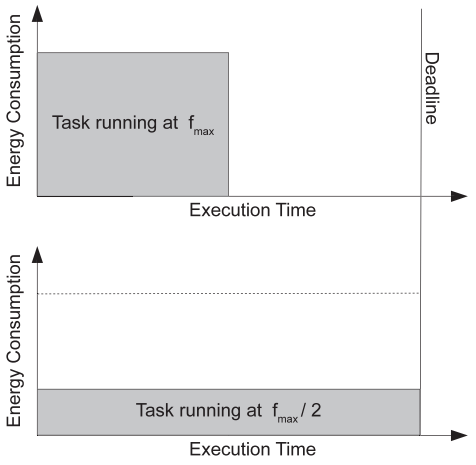

Fig. 2. This figure shows two executions of a task running at different voltage and frequency (DVFS) levels. In the top graph, the task runs at full speed and finishes well in advance of its deadline. In the bottom graph, the processor frequency (and supply voltage) is reduced by half so that the task's deadline is just met, which leads to less energy consumption.

significant amount of time and so they could be turned off during inactive periods to save energy. Hardware and software manufacturers have agreed to create standards such as the ACPI (Advanced Configuration and Power Interface) [9] which introduces several modes of operation enabling to turn off some parts of the system such as processor cores, hard drives, or Ethernet cards. Contemporary devices offer different *power states*. As an illustrative example, Fig. 1 depicts the different power states for processors implementing the ACPI specification. The idea behind DPM is that these devices periodically enter idle periods where no work is available and as a result can be brought into a less energy-hungry sleep state. The drawback is that when work becomes available the device needs to be brought back into the operational state, and this transition incurs additional latency. The lower the power state (higher number associated with it), the greater the energy savings; however, the transition latency also becomes higher for lower active states.

While DPM brings components into nonoperational but energy-efficient states, Dynamic Voltage and Frequency Scaling (DVFS) lowers the dynamic power used by the processor by limiting its clock frequency (f) and/or the supply voltage (V_{dd}). As stated earlier, a joint reduction in f and V_{dd} enables to reduce dynamic power cubically and energy quadratically but may lead to an increased execution time.

3.2 Algorithms Using DVFS and DPM

Weiser et al. [10] were the first to propose the use of DVFS to reduce the energy consumption in computing systems. They explored a set of OS-level scheduling algorithms to reduce CPU idle times by dynamically adjusting the DVFS level and evaluated each algorithm by means of simulated execution traces.

Early research efforts exploiting DVFS to reduce energy consumption arose in the context of real-time systems, where tasks' deadlines are given and their WCETs (*Worst-Case Execution Times*) are typically known beforehand. Many algorithms proposed in this domain have leveraged the *performance slack* available in real-time applications [11], [12], [13], [14]. The common approach is to minimize energy consumption by reducing the performance of the processor (DVFS level) but without causing the applications in the workload to miss their deadlines (as depicted by Fig. 2). This approach relies on the observation that completing an application before its deadline and then idling (e.g., via DPM), is less energy efficient than running the task at a lower clock speed and finishing it just in time. Yao et al. [11] propose an optimal static (offline) scheduling algorithm for independent tasks. Another static scheduling algorithm is presented by Ishihara and Yasuura in [12] which relies on the fact that a task's average switch activity (the $C_l \cdot N_{sw}$ factor in (1)) is known prior to the execution. Exploiting this additional information enables the proposed algorithm to reduce energy consumption by 70 percent. Aydin et al. [13] show that using exclusively WCETs to guarantee the temporal constraints of the workload, lack the potential to take advantage of the unused computation time since actual and worst-case execution times in real-time applications may differ significantly. Their experimental evaluation demonstrates that the proposed dynamic and speculative solutions, which exploit the unused computation time, achieve up to 50 percent energy reduction with respect to a static algorithm (based on WCETs). The two scheduling algorithms proposed by Zhu et al. [14] (designed for independent and dependent tasks, respectively) aim to overcome the main limitation of previously proposed techniques [11], [12], [13]: they were specifically designed for uniprocessor systems. As Aydin's proposals [13] the scheduling algorithms presented in [14] also exploit the unused computation time to dynamically adjust the processor speed but are capable of sharing this *temporal slack* among processors.

Another early work utilizing DVFS for dynamic power management was from Isci et al. [15]. Unlike more recent works in this area, they did not account for different effects of frequency scaling on different applications. Instead, they use a static estimate of power consumption and performance at each processor operating mode. They assume three operating modes: *Turbo*, where the processor runs at the highest

available frequency and consumes most power, *Eff1*, where voltage and frequency are reduced by 5 percent, and corresponding power savings and performance loss are statically estimated at 14 and 5 percent, respectively, and *Eff2*, with 15 percent voltage/frequency scaling, and 38 and 17 percent estimated power savings and performance loss. Their best power management policy, called MaxBIPS, simply chooses the mode that is expected to deliver the highest throughput (measured in Billions of Instructions per Second (BIPS)) given a power budget, based on the static estimate of power savings and performance loss in each mode. Although this simple policy delivered significant power savings, subsequent research attempted to take into account varying application characteristics when designing power management policies.

Most of the newer algorithms rely on the observation that different applications respond differently to changes in DVFS level, some suffering more performance loss than others. In particular, the work by Dhiman and Rosing [16] as well as many others [17], [18], [19] illustrate that the degree of performance degradation that an application experiences as the processor frequency is lowered depends on how compute-bound the application is. Tasks that are completely compute-bound, referred to as *burn_loop* [16], [20], show a linear dependence between processor frequency of execution time. Tasks that are completely memory-bound, referred to as *mem* [16], [20], show negligible slow-down as processor frequency is reduced. Finally, applications in-between the two extremes, termed *combo* [16], [20], have behavior in the middle. Memory-bound tasks suffer less from reduced CPU frequency, because the frequency of memory is not reduced. Choosing the right DVFS setting for combo applications is a delicate balance between reducing power consumption and increasing execution time.

Dhiman and Rosing [16] address this challenge by characterizing applications with respect to their degree of memory-boundedness. To do this online, they estimate each thread's *Cycles Per Instruction (CPI) stack* using hardware performance counters available on modern processors. CPI stack is a breakdown of cycles spent executing instructions versus stalling the processor on cache misses, TLB misses, and memory accesses. Each thread is then characterized by the metric μ, which is ratio of the cycles it spends on instruction execution to the overall cycles. A lower μ indicates memory boundedness, and higher μ (approaching 1) indicates CPU boundedness.

The authors then dynamically choose between several voltage-frequency settings, which they deem *experts*, depending on the μ-value prevalent for the currently executing workload. If the workload is predominantly CPU bound an expert with a higher voltage-frequency value would be chosen and vice versa. The μ for each task is continuously estimated to account for phase changes in programs. In relating the dynamically changing properties of currently executing workload to the system voltage-frequency setting, the authors achieve as much as 49 percent energy savings.

In [21], Lee and Zomaya exploit DVFS to devise energy-conscious task scheduling algorithms for heterogeneous distributed systems (multiprocessor and multicomputer systems). This complementary study analyzes DVFS techniques in the context of precedence-constrained parallel applications. They focus on exploiting CPU idle time slots caused by intertask communications and are able to reduce energy consumption with little effect on the overall execution times.

In [20], Dhiman and Rosing build on their DVFS-based power management algorithm [16] and on their earlier work that used purely DPM [22]. For each task they employ the techniques of [16] to select the best DVFS policy. However, when the processor becomes idle they use an approach similar to their dynamic DVFS algorithm to so decide if/when to put a processor into a lower power state.

Work such as [15] implicitly raises an important question about the granularity of DVFS control in a CMP. Is it necessary, for performance-energy optimization, to provide a per-core frequency and voltage control knob? Doing so adds significant complexity to the manufacturing of the chip while, on the other hand, offers unparalleled performance optimization potential. Herbert and Marculescu [23] attempted to answer this question by simulating a CMP with different granularities of DVFS control, which they term *VF-Islands* (each island has a voltage-frequency knob associated with it; by varying the number of cores found in an island they vary the control granularity). Their experiments with different workloads and DVFS policies show that providing per-core control will translate into minimal performance gains while dramatically increase scheduling complexity as compared to the control-granularity of several cores per VF-Island. This is also the dominant trend in modern architectures that place DVFS control at the granularity of a group of several cores,[3] most commonly the group of cores in the chip that share the last-level cache.

As mentioned earlier, DVFS is only capable to reduce the dynamic component of power consumption, which depends on f and V_{dd}. The increasing portion of static power in the total power dissipation of upcoming manufacturing technologies has led several researchers to explore the real limits of using DVFS to reduce energy consumption today. Cho and Melhem [26], [27] derive analytical models to study the potential of DPM and DVFS to reduce energy consumption for parallelizable applications on multicore systems. In their analysis, they consider two different scenarios: one where just DVFS is supported and the other where both DPM and DVFS are available. The obtained models reveal that substantially greater power savings can be obtained in the second scenario, where individual processors can be turned off. LeSueur and Heiser [28] take this observation one step further by illustrating that the energy savings benefits of DVFS are diminishing in recent processors due to rising static power consumption and reduced dynamic power range. Furthermore, their analysis suggests that in future generations turning-off unused processors (by setting these in ultra low-power states) will enable greater energy savings than using DVFS.

3. This is the case of cutting-edge Intel processors such as the Intel Core i7 [24] processor and the Single-chip Cloud Computer (SCC) [25]. An example of processor supporting core-level DVFS is the AMD Opteron "Barcelona" processor. This processor allows each core to operate at a different frequency, but the voltage must be no lower than that required by the core operating at the highest frequency.

3.3 Mapping Execution to Physical Resources

The work discussed thus far in this section has treated all computational cores as being completely equivalent and as such has not addressed the issue of mapping execution to physical contexts. However, due to the facts that different cores can be throttled to different degrees creating an artificially asymmetric system, and because variations in manufacturing create homogeneous cores which are not all that homogeneous, mapping of execution contexts to physical resources becomes a vital part of energy-cognizant scheduling.

Teodorescu and Torrellas [29] discuss scheduling in light of the fact that due to variations in the manufacturing process even homogeneous cores of a CMP are in fact heterogeneous. More specifically, the cores vary in the amount of power that they consume and in the maximum frequency that they support [29]. Work that focuses on intentionally manufactured asymmetric systems is discussed in Section 5. The authors of [29] explore algorithms that reduce power consumption in the presence of heterogeneity, such as first utilizing the least power-hungry cores, and mapping the most power-intensive threads to the least power-hungry cores. They also explore algorithms that improve performance in the presence of core frequency variation such as first utilizing cores that support the highest frequency or mapping high-IPC threads to high-frequency cores. Algorithms that optimize performance under the constraint of a power budget are explored as well. Having first mapped high-IPC threads to high-frequency cores they meet power budget constraints by applying DVFS to cores in a round-robin fashion, using a global linear optimization solution or simulated annealing.

Shin and Kim [30] consider the problem of constructing conditional task graphs for real-time applications, mapping the tasks to the available cores, and setting the DVFS for each core to just meet the timing deadlines while consuming the minimal amount of power. Curtis-Maury et al. [31] propose a multithreading library that uses online performance counters as well as a sophisticated performance/energy prediction model to decide on the optimal number of threads to run as well as the number of threads to map to each core to optimize the energy-delay product (EDP) for every phase of execution.

Kadayif et al. [32] take an interesting approach to the mapping/energy optimization problem. The authors make the observation that when the iterations of a nested loop are parallelized to be executed by multiple threads on multiple cores the amount of work that the different threads have to do can vary significantly. To demonstrate this issue consider a nested loop where the outer *for loop* iterates $(1 < j < 1{,}000)$ and the inner *for loop* iterates $(j < k < 1{,}000)$. Suppose this loop is divided between two threads such that all iterations with $j < 500$ are done by thread A and all other iterations are done by thread B. Because the inner loop is conditional on the value of j thread A has nearly twice as much work as thread B. In order to keep synchronization simple it would typically be the case that no further work will be scheduled until all the threads executing the same loop have finished. As a result, the core that executes thread B will be idle while thread A finishes. To compensate for this wasteful scenario the authors propose compile-time support that will estimate the workload assigned to each core (in a parallelization of nested loops scenario) and apply DVFS to the cores with less work assigned to them such that all threads will finish at roughly the same time. This makes each core consume the minimal energy possible to get its portion of the loop done, while keeping the execution time for the whole loop unchanged.

3.4 Integrating Contention Awareness into Energy/Power Solutions

Shared resource contention is a serious and well documented problem in chip multicore processors [33]. The issue stems from the fact that the cores of the CMP are not fully independent processors but rather share resources such as the Last Level Cache (LLC), prefetching hardware, the memory bus and DRAM memory controllers. Some researchers have measured that threads can slow down by hundreds of percent when competing for these resources with other threads on neighboring cores as compared to running alone on the machine. There has also been significant research effort on mitigating the negative effects of shared resource contention. Solutions ranging from hardware-based cache partitioning or DRAM scheduling to contention-aware OS thread schedulers have been explored. In this section, we survey energy-cognizant scheduling solutions that take into account the negative effects of shared resource contention.

There have been three distinct branches of research on the issue of shared resource contention in CMPs in the context of power/energy scheduling. Kondo et al. [34] use DVFS to mitigate shared resource contention and provide fairness. Takagi et al. [35] and Watanabe et al. [36] show how combining contention-mitigation techniques with power/energy scheduling leads to better solutions. Dhiman et al. [37] show that using purely contention-mitigation techniques also significantly reduces energy consumption.

The work by Kondo et al. [34] addresses the problem of shared resource contention, in particular memory bus contention, affecting different threads differently and hence distorting fairness. The authors define a thread's performance degradation as the ratio of instructions per second (IPS) with contention versus the IPS without contention. They define the unfairness between two threads as the difference in the performance degradation of these two threads. With the goal of minimizing the unfairness between threads sharing resources they employ the energy/power minimizing technique of DVFS to throttle the core running the more aggressive thread. In the proposed scheme, on every cycle they record the number of outstanding cache misses for every core, as well as the IDs of the cores whose misses are being serviced in that cycle. Using this information, the authors calculate the number of cycles that each core (with the corresponding thread on it) has delayed every other core. If the delay value between two cores is larger than a predetermined threshold then the offending core is throttled (using DVFS) to lessen its ability to impede the progress of others. By raising and lowering the frequency-voltage settings for cores in response to biased usage of shared-resources the authors improve fairness in a CMP.

While Kondo et al. [34] use DVFS as a tool to improve fairness, Takagi et al. [35] and Watanabe et al. [36] argue

that in order to use DVFS effectively as a power reduction technique on a CMP, shared resource contention must be first controlled by other means. If one of the threads gains greater access to the shared resources then it becomes necessary to compensate the other thread with a higher DVFS setting so that it still meets its performance deadlines. The power model derived in [36] reveals that the power consumption of the entire CMP is minimized if the frequency setting for every core is equal. In order to achieve this, the authors implement a priority control mechanism to access the shared memory bus. Power optimization is then achieved by controlling both the priority of memory accesses of the cores as well as the corresponding DVFS levels. The work by Takagi et al. [35] expands on [36] by also considering shared LLC contention. Along with priority control for the memory bus and setting DVFS such that the deadline is just met, [35] also partition the LLC among the competing threads based on stack distance profiles for these threads. Stack distance profiles are a compact summary of a thread's memory reuse pattern that can be used to predict the number of cache misses and therefore the expected performance of the thread with different amounts of cache space available to it. The downside of stack distance profiles is that, they cannot be obtained dynamically online for the majority of commercially available processors.

The work presented by Dhiman et al. [37] is highly correlated to work done on contention-aware CMP thread level schedulers. Studies such as [33], [38], [39], [40], [41], [42], [43], [44], [45] have used a myriad of different techniques to decide which thread-to-core mappings minimize shared resource contention. A common conclusion of these studies on contention-aware scheduling, was that a heterogeneous workload, one consisting of both compute- and memory-bound applications, is found to perform the best if different types of applications are paired together as opposed to when the same type of applications are paired together. Dhiman et al. achieve similar conclusions but stands out in that the stated goal of their work [37] was power/energy minimization. The biggest difference is that while the schemes presented in [33], [38], [39], [40], [41], [42], [43], [44], [45] separated applications in order to achieve higher performance, the algorithm proposed by Dhiman et al. [37] does so to minimize power consumption. The other noteworthy difference is that Dhiman et al. [37] consider not individual applications being mapped to cores, as was done in all the contention-aware scheduling studies cited, but rather Virtual Machines (VMs) being mapped to Physical Machine (PMs). However, since the authors only execute one benchmark per VM and the benchmarks are from the same benchmark suites that were used by the majority of the contention-mitigation studies, the results are similar. The authors found that when memory-intensive applications (VMs) are executed on the same PM performance significantly degrades resulting in longer execution times, which translates into higher energy use. The finding that memory-intensive applications suffer performance degradations and prolonged runtimes matches the conclusions reached in [33], [38], [39], [40], [41], [42], [43], [44], [45]. Another notable observation is that PMs loaded with compute-bound VMs will draw significantly more power (since compute bound applications are more power-hungry) than PMs loaded with memory-bound VMs leading to the undesirable effect of a power imbalance between the PMs. Dhiman et al. [37] propose a VM management utility called *vGreen* which schedules the VMs to the PMs in such a way as to balance the MPC (misses per cycle), IPC, and CPU-utilization across the PMs. Doing such balancing, which boils down to scheduling a heterogeneous mix of compute- and memory-bound VMs to each PM, resolves the power imbalance issue but more importantly reduces shared resource contention between the VMs facilitating a faster execution time and a reduced energy usage.

3.5 Summary

The manufacturers of modern processors are well aware of the need and desire to reduce power consumption of computing tasks as evidenced by the complex logic present ubiquitously on contemporary architectures to bring components into a low power state (DPM) or throttle processing power (DVFS). Ironically this logic, although available, often goes unused since hardware capabilities alone are insufficient and need to be paired with a decision mechanism which will determine when and by how much to invoke these features. The OS scheduler is a natural location to place such a decision mechanism as supported by the work surveyed in this section. The most obvious task that the scheduler must perform is to decide which components can be throttled and to what extent. One example is throttling cores running threads which are constantly blocked waiting on memory access. Such threads are prime candidates for throttling since they experience minimal slow downs at lower frequencies and hence offer pure energy savings. Complexity is encountered because the scheduler must be able to measure the memory usage characteristics of threads in order to apply this heuristic as well as deal with situations where threads are only partially memory-bound. Significant further complexity arises because cores, even in explicitly symmetric systems, are not necessarily homogeneous. Variations in the manufacturing process lead to cores with different power-dissipation properties and different maximum frequencies that they can support. This provides the energy-cognizant scheduler with the additional task of mapping contexts of execution to physical resources based on the properties of these physical resources on top of the original task of throttling the physical resources. Yet further complexity is added to the problem because of shared resource competition among threads scheduled to "neighboring" cores of a multicore processor. Shared resource contention can lead to significant performance losses which will directly translate into unnecessary energy expenditures. As such, a comprehensive energy-cognizant scheduler must be shared-resource aware and find a thread-to-core mapping as well as DVFS settings for these cores which minimizes shared-resource contention, accounts for the potential heterogeneity of cores, and satisfies some energy-performance metric.

4 Thermal-Aware Scheduling

Computer chips like any other manufactured good have a temperature range within which they were designed to

operate. When that range is exceeded the materials are stressed beyond their tolerance point and can melt, break, burn, or otherwise stop working resulting in a partial or complete failure of the chip. Such situations are called catastrophic failures and modern processors have technologies built into them, like AMD's Multipoint Thermal Control or Intel's Adaptive Thermal Monitor, to avoid these occurrences. These failure-avoiding techniques can be broadly described as *reactive* thermal management. The processor is designed to handle the heat produced by a "typical workload"; however, certain workloads can more aggressively use the processor resources generating more heat than can be dissipated, leading to dangerous overheating. These situations are detected via temperature probes strategically located within the chip and when the temperature exceeds a manufacturer-predetermined threshold the thermal management techniques are activated. Thermal management techniques can lower the frequency and voltage of the processor, apply clock gating (duty cycles) to the processor, or halt it altogether. This is done until the temperature falls back below the acceptable threshold, at which point normal operation is resumed. Because these techniques are activated only once a temperature-critical situation has been reached and because they do nothing to prevent reaching such situations, they are classified as reactive techniques.

The nature of reactive thermal-management techniques that lower temperature by reducing processor capabilities has the potential, if active for long durations, to have a significantly adverse effect on thread performance. Much research has been done on *proactive* thermal management that takes preventative measures, which themselves have minimal or no performance impact, ensuring that critical temperatures that would trigger costly hardware reactions are never reached. Proactive thermal management relies on the old adage "a stitch in time saves nine."

Thermal crises leading to component failure are the most dramatic but not the only examples motivating the need for thermal management. Temperature gradients can be created in both space and time; they will stress the underlying materials and dramatically decrease the expected lifespan of the device [46]. Temperature gradients in space arise because different threads heat the processor differently. A thread that is very compute-bound will heat the processor by constantly issuing instructions, much more than a memory-bound thread, which will be stalled for most of its execution time. If such different threads are scheduled on adjacent cores then the resulting temperature gradient will be harmful to the chip's long-term durability. Similarly, if a core is frequently switched between hot and cold threads then the resulting temporal temperature gradient will not be good for the components either. Proactive thermal management takes these effects into account as it tries to avoid hot spots and temperature gradients, by distributing the temperature as evenly across the chip as possible. Furthermore, temperature is directly proportional to leakage current, and so techniques that minimize the overall temperature are highly beneficial in terms of energy/power savings as well.

In this section, the proactive techniques are further broken down into *static techniques* that determine the scheduling solution prior to the start of execution, and *dynamic techniques* that work throughout the execution of the workload to make improvements. Dynamic techniques are further divided into those that use temperature measurements directly available from on-chip sensors and those that construct thermal models to predict future temperature data in order to make policy decisions. We conclude the section with a discussion of how different thermal management strategies can be integrated in order to provide more robust solutions.

4.1 Static Thermal Management

One subset of proactive thermal-aware scheduling algorithms relies on static scheduling to meet predefined thermal goals. The static scheduling problem is best described by breaking an application into tasks, where each task is a function or a set of functions. The entire application is then represented as a Directed Acyclic Graph (DAG) of computation with the vertices representing the tasks and the directed edges representing the precedence relationships between the tasks. The problem becomes scheduling these tasks to the available cores such that the timing constraints for the tasks are met, the precedence constraints are met, as well as some thermal goal, such as keeping the max temperature below a particular threshold, is also met. Since the scheduling decisions are made once prior to the start of the run rather than dynamically during execution, these solutions are called static. Different types of algorithmic solutions have been proposed to tackle this problem.

Zhang and Chatha [47] derive an optimal solution to the static thermal-aware scheduling problem. They show that the problem of choosing a frequency at which to run each task as well as choosing the length of the low-power period for the processor in-between tasks such that the predefined temperature threshold is not exceeded and that the entire workload completes as quickly as possible is NP-hard. They show a *dynamic programming* (DP) optimal solution that runs in pseudo-polynomial time and an approximate fully polynomial time solution to this problem. Coskun et al. [48] use integer linear programming (ILP) to find scheduling solutions (mappings of tasks to cores as well as choosing each task's frequency/voltage setting) such that one of the following four metrics is optimized: the minimization of hot-spots (where one core or portion thereof significantly exceeds average chip temperatures), the minimization of hot spots and thermal gradients (where there is a large temperature discrepancy between on chip resources), balancing energy load across the cores, or minimizing the total energy consumed.

Chantem et al. [49] use mixed integer linear programming (MILP) to decide on a mapping of tasks to a floorplan of cores as well as an ordering of tasks on each core such that all precedence and timing constraints are met while the preset max temperature is not exceeded. It is a well established fact that the rate at which cores cool depends on their location within the floorplan of the die. If cores are near edges they can more easily dissipate heat via those edges than cores which are in the middle of the chip. Chantem et al. use the cores' positions within the floorplan as part of its thermal-aware solution [49].

4.2 Nonpredictive Proactive Thermal Management

The next subset of thermal-aware scheduling solutions that we consider is dynamic: they are active throughout the execution of the workload. These solutions are proactive since they attempt to facilitate favorable thermal conditions in the future. They are nonpredictive in a sense that they rely on temperature measurements/estimations of the current chip state and do not create thermal models to forecast future temperatures.

Although modern chips come equipped with thermal sensors that can be read by software, these readings are subject to noise and other reliability issues; furthermore, the sensors are limited to only certain locations within the chip [50]. As such, preprocessing temperature readings prior to utilizing them in any scheduler can increase its accuracy and effectiveness. Sharifi et al. [50] propose a scheme to make online temperature measurements significantly more accurate. Offline they construct a thermal equivalent RC (Resistor-Capacitor) model for the circuit and reduce its complexity to make the model manageable. They apply Kalman filtering calibrated with sensor readings until a steady state is reached. The model is then usable online to convert the unreliable sensor data into reliable thermal readings.

Choi et al. [51] propose a set of hot-spot mitigation techniques incorporated into the OS scheduler and evaluate the effectiveness of these techniques on an IBM Power 5 processor. The foundation of their proposals lies on two observations: 1) functional units' power and temperature profiles are tightly related to per-unit utilization, in view of advanced fine-grained clock gating technologies incorporated into modern processors, and 2) because the OS is already performing thread scheduling with a time granularity much smaller than rise- and fall-times on-chip temperatures, thermal mitigation techniques can be implemented at the OS level with low overhead. Their experimental evaluation shows that due to advanced clock gating in their target processor, exploiting the temporal and spatial heat slack when making scheduling decisions has a significant effect on the processor's temperature.

One of the key tools in proactive thermal management is the movement of threads from one core to another; usually from a hot core to a colder core. However, if migration of threads is explicitly disallowed, the throttling strategy proposed by Rajan and Yu [52] is mathematically proven to maximize throughput while maintaining the core temperature below the predefined threshold. The strategy throttles the core via DVFS to a precalculated level if the threshold temperature is reached. When the temperature falls below the threshold, the core is throttled back up to some maximum level that will not exceed the threshold. This is shown to be optimal for maximizing throughput over any other strategy so long as migration is disallowed. Stavrou and Trancoso [53] propose thermal-aware scheduling techniques that, although not explicitly restricting migration, focus on assigning newly dispatched threads onto cores rather than moving threads between cores. They propose scheduling techniques that assign the newly spawned thread to the coolest core on the chip. A more sophisticated version looks not only at the temperature of individual cores but also at the temperature of cores in the *neighborhood* around each core as well as the floorplan position of the core and computes a simple thermal goodness metric for each core. Further sophistication is added to this method by restricting scheduling to cores that exceed a predefined temperature threshold.

Coskun et al. [54] compare scheduling policies very similar to those proposed in [53] (e.g., mainly sending ready threads to the coolest core and sending ready threads to a core which "lives in the coolest neighborhood") against a more sophisticated technique called *Adaptive Random*. Adaptive Random assigns a probability of sending a thread to a core based on that core's past thermal history. A sliding window approach is used to keep track of the core's thermal history. At the end of each measuring period, the core's probability of receiving a thread is incremented if its temperature never exceeded a predefined threshold and decremented otherwise. The destination core for a thread is decided by generating a random number and determining the core to whose id that random number corresponds. The core's probability value as maintained via the sliding window method directly determines its likelihood of being the recipient of the thread. Thus, cores with a better thermal history (those that spent more time below the threshold temperature) are more likely to be assigned threads.

Almost ubiquitously, thermal management algorithms try to ensure that the core is heated as little as possible while still performing an acceptable amount of work. Gomaa et al. [55] propose a strategy which is radically unique in that it actually seeks to more aggressively heat the cores. Although initially such an idea may seem counterintuitive there is a beautiful logic behind it: the authors argue that whether only one component of the core becomes overheated or 10 components become overheated, the time for the core to cool will be roughly the same whether regardless of the number of overheated components. However, the workload that heated 10 components within the core rather than only one most likely got significantly more work done and yet suffered the same cooling penalty as the less efficient workload. Gomaa put this radical theory into practice using a hyper-threading enabled CMP [55]. The proposed algorithm has two parts *Heat and Run Thread Assignment* (HRTA) and *Heat and Run Thread Migration* (HRTM). HRTA is responsible for pairing threads onto the same core with complementary resource demands such that they will heat different components within the core. The authors attempt to pair threads based on two main criteria: integer with floating point, and high IPC with low IPC. If thread pairings based on these broad criteria are not available then HRTA looks at more specific resource usage such as the cache and the ALU. They also seek to balance the IPC evenly across all cores to lengthen the times between cooling periods. Once a core has reached its threshold temperature, HRTM is responsible for migrating threads away from the overheated cores created by HRTA, allowing them to cool. It should be noted that the scheme proposed in [55] works only if the number of threads is smaller than the number of thread contexts available in the CMP, since HRTM relies on the existence of cooler, not overloaded, cores to migrate threads from the hot overloaded cores.

The work by Zhou et al. [56] analyzes thermal management issues on 3D processors, which consist of several

vertically stacked dies coupled with a dense, high-speed interface. Recent research has shown that 3D die stacking is an exciting new technology that increases transistor density and, at the same time, enables a significant reduction of interconnect both within a die and across dies in a system [57], [58]. In spite of these benefits, 3D die stacking gives rise to an increased power density per unit volume. A novel scheduling algorithm is proposed by Zhou et al. [56] to address this issue by taking into account the thermal conduction in the vertical direction. Based on the observation that a core in one layer could become hot because of a high-power task running on the same vertical column but a different layer, the proposed scheduler performs thread-to-core assignments for sets of cores that are vertically aligned (*super cores*) rather than based on the temperatures of individual cores. Their experimental evaluation reveals that this scheduling technique significantly reduces the number of thermal crises and, so, it delivers better performance than conventional techniques designed for 2D processors. Furthermore, the authors argue that hardware techniques aimed to dramatically reduce a core's temperature upon a thermal emergency must be engaged in a per-core-stack fashion, and more specifically must be targeted to the core that contributes more significantly to increasing the temperature in this stack.

4.3 Predictive Proactive Thermal Management

This next subset of thermal-aware scheduling techniques is similar in both goals and strategies to the previous subset except that instead of relying on temperature measurements from on-chip thermal sensors to drive decisions these management techniques incorporate sophisticated models to predict future core temperatures. The downside of this approach is that it adds significant complexity for the implementation both in terms of the code needed and the calculation time. The upside is that the scheduler becomes significantly more powerful being able to anticipate thermal crises as well as take actions that not only mitigate problems observed at the moment, but also reduce the probability of future crises.

Work by Yeo et al. [59] is a perfect example of how temperature predicting models can extend the thermal-aware policies described earlier. The authors create a two-phase prediction model on the observation that the rate of change of temperature for an application is proportional to the difference between its current temperature and its steady-state temperature (the temperature that it would acquire if running in a continuous loop). The two phases correspond to the *Application-Based Thermal Model* (ABTM) and the *Core-Based Thermal Model* (CBTM). The ABTM predicts the thread's future temperature using a recursive least squares method while CBTM relies on a simple differential equation, the core's initial temperature, the application's steady-state temperature, and the observation above to predict the core's future temperature. The overall predicted future temperature for the thread running on the core is computed as a weighted sum of the predicted thread and predicted core temperatures. The obtained predicted temperatures are then used to perform thermal management. More concretely, if the predicted temperature of a certain thread-core-combination exceeds the defined threshold then that thread will be migrated to what is predicted to be the coolest core. It is easy to see how there is a clear advantage in performing such a migration without having to actually reach the threshold temperature.

A much more involved technique called *Autoregressive Moving Average* (ARMA) is used by Coskun et al. [60], [61], [62] to predict future temperatures. Under the assumption that temperature follows a stable distribution with a fixed mean and variance the authors model variations in temperature as a stochastic process. Offline they fit the collected data to a progressively higher order model until the error falls below a specified threshold thus obtaining the model parameters and residuals. The model residuals are checked for randomness ensuring no autocorrelation. An online adaptation is used to ensure that the model still describes the current workload. Workload changes are detected by applying a *sequential probability ratio test* (SPRT), i.e., statistical hypothesis testing on the distribution of model residuals. The model is adjusted on the fly to compensate for workload changes. The ARMA model allows thread temperatures to be forecasted into the future. The authors explore using the predicted thread temperatures in several thermal-aware scheduling algorithms. Proactive-Migration moves threads away from cores that are predicted to become excessively hot. Proactive-DVFS reduces the frequency/voltage of a core if its temperature is predicted to exceed a certain threshold. Finally, *Proactive-Thread Balance* (PTB) uses the ARMA results to migrate threads between the run queues of different cores to even out the temperature across the chip.

Although ARMA has been shown to be effective, its complexity makes it an unattractive option for an online implementation. Azoub and Rosing [63] propose a more cost-effective prediction model based on the band limited nature of the temperature frequency spectrum. The proposed *Band-Limited Predictor* (BLP) requires no training phase and is able to accurately predict future events based on past events for a band limited signal (one which takes values only from a known range). The prediction results from this model are used in a similar fashion to those of other models mainly to migrate threads from cores predicted to exceed the temperature threshold to those predicted to be cold. The model is also used to allocate newly spawned threads to cores predicted to be cold.

Kumar et al. [64] propose a thermal model that differs from the others in this section in that it does not seek to predict the future temperature of threads but rather to separate the temperature contributions made by the different threads time-sharing the same core. The approach relies on observing how much each thread uses various processor resources and correlating that to a temperature metric for those resources. More specifically, for each of the threads 22 performance metrics are dynamically collected online corresponding to the activity of different micro-architectural components of the processor. Offline partial least squares regression analysis is used to generate the coefficients necessary to convert the performance counter values into temperate values. The model is periodically, in a limited way, adjusted online during execution. The model is used to dynamically predict the temperature of

individual threads on the core. If a thread's predicted temperature exceeds a predefined threshold then the proactive software module will use the priority mechanism available inside the Linux OS to limit the amount of time that thread runs on the core. The size of the time slices allocated by Linux to a particular thread when multiplexing a processor between different threads depends on the priority level of that thread. The smaller the priority the longer the time slice. By decreasing the priority of "hot" threads the authors ensure that they run less, lowering the overall temperature of the core. In the method proposed in [64], if the proactive adjustment of thread priorities does not prevent overall core temperature from surpassing the predefined threshold then a reactive hardware technique is triggered, which cools the core by clock modulation.

By means of a comprehensive experimental analysis, the work by Yang et al. [65] showcases two limitations of Kumar's approach [64]. First, given that hot jobs are executed less frequently than cool jobs, cool jobs typically finish earlier than hot jobs. Putting off the execution of hot jobs may lead to a high number of thermal emergencies when the cool jobs are exhausted, thus causing a significantly adverse effect on performance. Second, Kumar's scheduler is inherently unfair since it tends to allocate less CPU time to hot jobs and more to cool jobs. In [65], a scheduling algorithm that is not subject to these limitations is presented. The algorithm follows the strategy of keeping the temperature right below the threshold but without exceeding it, based on the observation that this approach increases the heat removal rate and is less likely to incur thermal emergencies. As the next task to run this scheduler selects the hottest program that does not increase the temperature above the threshold; if such job does not exist, the hottest job is selected to run. The authors derive an elaborated estimation model that makes it possible for the scheduler to accurately predict the future temperature that results from running a particular job in the next time interval.

4.4 Integrating Multiple Thermal-Management Techniques

In this section, we discussed a wide variety of thermal-aware scheduling techniques covering the available literature on the subject. Each technique was, under certain circumstances, shown to be effective. Significant evidence has also been presented showing why thermal management is a vitally important issue for contemporary and future CMPs. The question that may logically arise is which method is better and should be adopted in operating systems by default? However, such a question is not the right one to ask. These methods are not mutually exclusive choices. There are positives and negatives about each method. There are situations and workloads where some methods work well and others fail and different circumstances where the roles are reversed. A promising direction is to combine several thermal management techniques, combining their strengths.

Some exciting work has already been done in the direction of multimethod thermal management. The work by Kumar et al. [64], described earlier, uses a proactive software technique to reduce the time "hot" threads spend on the processor but also uses a reactive clock modulation approach to guard against situations where the first method is insufficient. By default all modern processors are equipped with hardware-based thermal crisis avoidance techniques like the Adaptive Thermal Monitor in Intel's Core i7 chips [24]. Thus, any implemented thermal management technique is also safeguarded in hardware. Coskun et al. [66] demonstrate how a static thermal management technique first described in [48] can be combined with the Adaptive Random dynamic thermal-management technique introduced in [54] to improve overall efficiency.

Coskun et al. propose a framework in [67] for combining a variety of thermal management techniques and choosing the most appropriate one for the given workload. They reduce the problem of choosing a thermal-management technique to the *switching experts problem*. The experts are thermal-management policies. The expert to use during any given period is decided by a specialist based on a set of rules the specialist has and the currently measured load on the system. The specialist to use during any given period is decided using an online learning method that evaluates the specialists based on their performance or would-be performance in the past. The experts considered in [67] were:

1. the default OS scheduler,
2. DPM, which turns off idle cores after a fixed timeout,
3. DVFS+DPM, which scales the frequency of cores based on their utilization level and turns off idle cores,
4. migration, which moves threads from hot cores to colder ones, and
5. Adaptive Random [54].

The two specialists considered were *utility based* and *temperature based* each of which supports a subset of experts and decides which expert to use depending on the total system load.

Donald and Martonosi [68] provide a comprehensive study of how different thermal management policies can be used together. They divide the field into three orthogonal axes. The first axis describes if the policy uses stop-go (clock modulation) or DVFS. The second axis describes if the policy is applied globally to the CMP as a whole or to each core individually. The third axis describes if a migration policy is used and if so whether it is based on performance counters or thermal sensors. All possible permutations of these thermal-management policies are explored and a matrix showing the throughput achieved without exceeding a temperature threshold is derived. A major conclusion of that study was the value of DVFS which showed a 2.5X speedup in throughput over stop-go on average.

4.5 Summary

Even the most complex computer chip can still be thought of as an expensive resistor since it converts electric current that passes through it into heat. Excessive heat can damage the chip and turn it into an actual resistor. To prevent this, modern hardware is equipped with built-in thermal crisis prevention technologies that detect that the temperature has gone above an acceptable threshold and reactively halt or throttle the hardware until the temperature falls below this threshold. Although effective at avoiding complete meltdowns this reactive technology has the potential to impede

performance and does nothing to prevent intrachip thermal gradients, which can significantly limit the expected lifetime of these components. Proactive thermal-management performed by the OS scheduler can address thermal crises before they happen minimizing performance effects as well as addressing issues like thermal gradients. If the workload is known a priori it is possible to perform static thermal scheduling. The workload is broken into tasks, with dependencies among them captured in a graph. A schedule is then created by deciding on which tasks will run on which cores, when and to what degree each of the cores will be throttled, such that performance is optimized while avoiding any negative thermal effects. Since it is typically not the case that the workload is known a priori less efficient but far more realistic dynamic proactive-thermal-aware scheduling becomes necessary. Using filtered or otherwise preprocessed readings from temperature probes located on the chip, often in conjunction with other data, such as the floorplan of cores or thread-performance counters the scheduler decides on thread-to-core mappings and core throttling to deliver peak performance without negative thermal effects. Further effectiveness but also complexity can be added to dynamic thermal-aware scheduling by constructing thermal models to predict future core temperatures and using these take truly proactive actions.

5 Scheduling for Asymmetric Systems

An asymmetric multicore system is defined to include several cores that expose the same instruction-set architecture (ISA), but offer different performance, occupy different size on chip areas and have different power characteristics. Systems may be built asymmetric by design [69]—we refer to these as *explicitly asymmetric*—or asymmetry may arise as a result of nonuniform frequency scaling on different cores, or if variations in fabrication process prevent all cores from running at the same speed [70]. Explicitly asymmetric systems are also referred to as single-ISA heterogeneous systems. There is also another type of heterogeneous systems, where cores expose different ISA: for example, the IBM Cell processor has one PPE (a general-purpose core) and a number of SPEs (special-purpose vector processing units). These systems require primarily compiler support, because code must be explicitly compiled to run on a particular core type. As a result, scheduling for these systems has not been an active area of research, at least at the time of this writing. Instead, we focus on asymmetric systems with a uniform ISA. On these systems, a thread scheduler is essential for maximizing system efficiency.

In explicitly asymmetric systems cores differ in microarchitecture. There is typically a small number of fast and powerful cores with complex microarchitectural features (super-scalar out-of-order pipeline, aggressive prefetching, and branch prediction hardware, high clock speed), as well as a large number of slower and low power cores. The latter are characterized by simpler pipelines, slower clock frequencies, smaller areas, and lower power consumption as compared to the fast cores. The motivation for this architecture is superior performance per watt that can be achieved if the microarchitecture of a core is tailored to the workload. The idea is that fast cores are better suited for 1) workloads with high instruction-level parallelism (ILP) that effectively utilize the fast cores' advanced features, and 2) sequential applications or sequential phases of parallel applications. High-ILP workloads, to which we refer as *CPU intensive* following the terminology adopted in earlier studies, achieve a greater speedup on fast cores relative to slow cores, because they utilize fast cores effectively. Low-ILP workloads, to which we refer as *memory intensive*, waste resources of the fast cores and typically achieve a better performance/watt when running on the slower cores. Scalable parallel applications can get good performance on slow cores, because they are able to spread the computation across them, while sequential or modestly parallel applications need to run on fast cores to get good performance. We refer to this notion of preferring certain types of cores for certain types of computation as *specialization*. To deliver specialization the system must match each instruction stream with the type of core best suited to this stream. This process is typically a part of thread assignment or scheduling. Therefore, it is typical to provide specialization by means of *asymmetry-aware thread schedulers*.

An asymmetry-aware scheduler would assign threads to cores so as to deliver specialization: threads that benefit most from fast cores would be preferentially assigned to run on fast cores, while other threads would be relegated to slow cores. Note that most schedulers for explicitly asymmetric systems assume cores of two types: fast and slow. Kumar et al. [69] showed that having only two core types is sufficient to deliver the projected benefits of asymmetric systems.

The remainder of this section is organized as follows: Sections 5.1 and 5.2 survey different scheduling algorithms aimed to optimize the overall system efficiency on AMPs for workloads consisting of single-threaded and multithreaded applications, respectively. Section 5.3 is devoted to exploring schedulers that exploit less conventional types of specialization (i.e., not directly related to the workload's ILP and parallelism). Section 5.4 focuses on asymmetry-aware schedulers pursuing goals other than optimizing for the overall system performance, such as fairness. Finally, Section 5.5 introduces another category of schedulers targeting AMPs where asymmetry was introduced as a result of variation in the fabrication process.

5.1 Addressing Single-Threaded Applications

The biggest challenge in designing an asymmetry-aware scheduler for multiapplication workloads consisting of single-threaded applications is to determine to what degree the throughput of individual processes (or threads) will improve when running on a fast core relative to a slow core. Throughput is usually measured in terms of instructions per second (IPS), and the relative improvement is often referred to as the *speedup factor* or *relative speedup*. Schedulers proposed by Kumar et al. [71] and Becchi and Crowley [72] used the most natural and straightforward approach for determining relative speedup. These schedulers simply ran each thread on each type of core and measured its IPS in both cases. Once IPS on both core types was known, the scheduler computed the ratio of fast-core IPS to slow-core IPS. Threads with a higher ratio would be

then assigned to run on fast cores and threads with a lower ratio would be assigned to slow cores. (Kumar also proposed another algorithm based on sampling of entire thread assignments—we will discuss this algorithm later.)

While this approach was intuitive and simple, it did not work well in practice. The problem was rooted in phase changes. If a thread changes its phase during the period when the scheduler performs a measurement of its IPS on fast and slow cores, the resulting IPS ratio may be estimated incorrectly. Suppose, for instance, that a thread had a naturally high IPS while it was running on a fast core, for example, because it was in a CPU-intensive phase. Then, when it was moved to a slow core it entered a very memory-intensive phase and so its IPS dropped. The scheduler would register a very high IPS on the fast core and a very low IPS on the slow core. The resulting IPS ratio would be high, signaling the scheduler that the thread should be assigned to the fast core. In reality though, the thread has entered a memory-bound phase and so assigning it to a fast core is counterproductive. (As we mentioned earlier, memory-bound threads use fast cores inefficiently, because they frequently stall the pipeline and make poor use of the advanced microarchitectural features.) Another problem with this algorithm was that scarce fast cores were in much higher demand than slow cores, because the scheduler often needed to remeasure the IPS of one thread or another on a fast core. This led to load imbalance and hurt performance.

These problems were first discovered by Shelepov et al. [18]. This group was the first to implement algorithms proposed by Kumar and Becchi on a real system. Kumar and Becchi evaluated their schedulers on a simulator, where it was assumed that an OS scheduler could measure IPS on different core types *instantaneously*. As a result, the problem related to phase changes did not surface in this simulated environment. Shelepov, on the other hand used a real system. Since actual asymmetric systems were not yet built at the time that the research surveyed herein was conducted,[4] asymmetry had to be emulated by setting cores of a symmetric machine to run at different frequencies. Kumar's and Becchi's decision to use a simulator was certainly well justified: they were able to model the microarchitectural details of asymmetric systems that researchers relying on frequency scaling could not. However, lack of realistic details in the model of the OS scheduler's actions concealed some problems with these algorithms.

Shelepov et al. addressed the aforementioned problems via a new concept of *architectural signatures* [18], [73]. An architectural signature is a compact summary of the program's architectural features. It can be obtained via binary instrumentation on any machine of a given architecture (e.g., x86) and then used on any system with that architecture, regardless of the differences in microarchitecture. The idea is that the architectural signature would be obtained during the development process and embedded in the application's binary. Then at runtime the scheduler would interpret this signature to *predict* each program's relative speedup on cores of different types. The signatures evaluated in Shelepov's paper were constructed using the stack-distance profile of the program [74], which could be used to estimate this program's cache misses for a last-level cache of arbitrary size and associativity. Predicted cache miss rates for a number of realistic cache configurations comprised the contents of the architectural signature. At runtime, the scheduler used the miss rate for the target architecture to predict the relative IPS of the thread on cores running at different frequencies. A scheduler relying on architectural signatures was immune to problems discussed earlier, because it did not need to run each thread on each core type to determine its IPS ratio. In fact, Shelepov et al. showed that a scheduler based on architectural signatures delivered better performance than a scheduler requiring dynamic IPS measurements.

While Shelepov's scheduler addressed the problems with earlier schedulers and resulted in simple and elegant OS implementation, it had limitations. The problem was that the architectural signature is inherently static. Therefore, it does not take into account dynamic behavior of the program, such as the variation in the IPS due to phase changes. Furthermore, it is difficult to make the signature account for differences in program inputs, which can be known only at runtime. Limitations of architectural signatures were overcome by other researchers, who found methods to determine relative speedup *without* needing to run threads on each core type and *without* relying on architectural signatures. The idea proposed by Saez et al. [17] was a straightforward and effective extension of Shelepov's model: the IPS ratio was predicted using miss rates that are measured dynamically on line, as opposed to relying on the miss rates delivered in the architectural signature. Miss rates can be measured on any core on which the thread runs, and so there is no need to sample performance on different cores. Based on this idea, Saez developed the *Speedup-Factor* driven and CAMP schedulers.

Koufaty et al. [75] concurrently with and independently from Saez et al. [17] developed a similar model. Their asymmetric system was different from that used by Saez. While Saez, like many researchers, emulated asymmetry by setting the cores of a real system to run at different frequencies, Koufaty et al. were able to configure their system such that one core had a smaller retirement width than another core. They were able to do this thanks to proprietary tools that were available to them at Intel. Koufaty also used hardware performance counters to determine the relative speedup, but unlike Saez's model, his model did not seek to predict the speedup precisely, but rather find performance metric that had a positive correlation with the speedup. On their experimental architecture, Koufaty determined that the rate of off-core requests (i.e., last-level cache accesses and misses) had a negative correlation with the speedup on the fast core. This finding is similar to that of Saez et al., who determined that a high last-level cache miss rate is a key factor in determining the relative speedup.

5.2 Addressing Multithreaded Applications

The schedulers discussed in the previous section addressed only a single type of specialization: where fast cores were

4. ARM announced its first asymmetric processor, combining Cortex-A7 and Cortex-A15 cores, known as the big.LITTLE configuration, in October 2012.

dedicated to programs that are likely to use those cores efficiently. Another group of schedulers addressed the second type of specialization: where fast cores were dedicated to sequential applications and to sequential phases of parallel applications. Hill and Marty demonstrated that when this specialization policy is followed, asymmetric systems are likely to always outperform symmetric systems as long as the length of parallel phase constitutes more than 5 percent of total execution [76]. To accomplish these performance benefits, several researchers designed schedulers that preferentially scheduled sequential applications and sequential phases of parallel applications on fast cores, while assigning threads running parallel phases to slow cores [77], [78]. Speedups resulting from this technique as compared to asymmetry-unaware schedulers reached the vicinity of 50 percent. The challenge in implementing these schedulers was to detect sequential phases in those applications that do not block their threads when they run out of work. To accomplish that, Saez et al. proposed a new system call interface allowing threads to notify the operating system when they enter an idle state [77].

Another challenge in implementing these schedulers was to mitigate migration costs, which could be quite substantial if threads switched phases often. Saez et al. showed that it is difficult to completely conceal migration overhead when fast and slow cores are located in different memory hierarchy domains (e.g., when they do not share the last-level cache). But if the asymmetric system is built such that fast cores share the memory-hierarchy domain with at least several slow cores, migration overheads become negligible [77].

5.3 Other Types of Specialization

While most asymmetry-aware schedulers were concerned with two types of specialization (related to the workload's ILP and parallelism), a few researchers looked at less conventional types of specialization. Mogul et al. proposed using slow cores in asymmetric systems for executing system calls [79], since system calls are dominated by code that uses fast and powerful cores inefficiently. They modified an operating system to switch the execution to a less powerful core when a thread enters the system call. This strategy had mixed results, but overall delivered better energy efficiency. Mogul also pointed out that workloads that execute OS-like code (e.g., code dominated by interrupts and frequent control transfers, such as web servers) are also good candidates for running on slow cores. Inspired by this idea, Kumar and Fedorova [80] proposed binding the controlling domain $dom0$ of a virtual machine monitor Xen to slow cores. They also observed that workloads dominated by activity in $dom0$ are less affected by variations in the speed of the core than other workloads.

5.4 Fairness and Performance Stability

Most asymmetry-aware schedulers were targeted at optimizing overall system efficiency. Nevertheless, there have been a few studies exploring additional challenges that arise in the context of AMP systems, such as delivering fairness or ensuring stable execution times.

One of the first schedulers addressing these problems was described by Li et al. [81]. In Li's scheduler fast cores received a higher load than slow cores. This was done to account for the fact that fast cores perform more work per unit of time. Li's scheduler did not take into account, however, that different threads experience different rates of speedup when running on a fast core relative to slow cores. As a result, under certain workloads it was possible to run into a situation where threads assigned to fast cores make slower progress than threads assigned to slow cores. Li's scheduler implicitly addressed fairness: threads running on fast cores would accomplish more work per unit of time than threads running on slow cores, but they would receive a smaller share of CPU time because fast cores were given a higher load. On the other hand, if the number of threads in the workload did not exceed the number of cores, Li's algorithm would not deliver fairness. To address fairness in this scenario, the scheduler needs to periodically rotate threads among fast and slow cores in a round-robin fashion. Such asymmetry-aware round-robin scheduler was described by Fedorova et al. [82] and in another article by Saez et al. [77]. Balakrishnan et al. described a simple asymmetry-aware scheduler that ensured that the fast core does not go idle before slow cores [83]. Despite its simplicity the scheduler significantly improved performance stability of realistic workloads. Kazempour et al. also addressed fairness and priorities on asymmetric systems, but their scheduler was implemented in a virtual machine hypervisor [84].

5.5 Addressing Process Variation

Another category of schedulers addressed systems where asymmetry was introduced as a result of variation in the fabrication process. Process variation causes cores to run at varying frequencies and consume varying amounts of power [70]. Therefore, the assumption that there are only two types of cores no longer applies. Other than that, the problem is similar to that on explicitly asymmetric systems: how to match threads to cores so as to minimize some global function, for example delay or energy/delay product.

Three algorithms addressing this problem were described by Winter and Albonesi [85]. Both algorithms relied on sampling performance of threads on each (or most) core types. The first algorithm, the Hungarian algorithm, was used to solve weighted bipartite matching problem and relies on an $N \times N$ matrix, where N is the number of cores and threads. An entry (i, j) in the matrix correspond to the cost of running a process i on core j. In this case, cost was represented by ED^2 (energy-delay-squared). Rows and columns of the matrix were manipulated according to the algorithm to find an assignment that minimized total ED^2. The second algorithm considered by Winter and Albonesi relied on global iterative search. The scheduler tried various randomly chosen thread assignments and then used the one with the lowest total cost. This strategy was similar to one of the algorithms proposed by Kumar et al. [71] for explicitly asymmetric systems. The final algorithm explored by Winter and Albonesi used local search. Local search is different from global search in that it probes assignments that are "close" to the initial one in the search space—for example, an assignment that can be derived from the initial one with a small number of swaps in the assignment of threads. The Hungrarian algorithm turned out to perform the best, but its complexity was $O(N^3)$ in the number of cores. This made it an unlikely candidate for systems with

hundreds or thousands of cores. Local search, on the other hand, had a lower complexity ($O(N)$), but performed similarly to the Hungarian algorithm.

Algorithms designed by Winter and Albonesi required sampling of threads' performance on different core types, and in particular the Hungarian algorithm required that each thread is sampled at each frequency. As other researchers have shown, this requirement may cause problems, such as incorrect speedup estimates and load imbalance [18]. Further, on systems where asymmetry is caused by process variation fabrication and the number of different core types may be substantially larger than two, sampling complexity further increases. An algorithm proposed by Ghiasi et al. [86] overcame this shortcoming. Their algorithm also targeted systems where cores have the same microarchitecture, but run at varying frequencies, but it did not require sampling of threads' performance on different cores to construct a good assignment of threads to cores. Instead, Ghiasi used a performance model to predict how a thread's IPC (instructions per cycle) is affected by a change of frequency. This performance model, similar to that of Koufaty et al. [75] and Saez et al. [17] was based on the number of off-core request issued by the thread. Ghiasi's algorithm was implemented and evaluated on a real system, and so it was built with practical considerations in mind: the algorithm was structured in a way that did not require examination of all possible mappings of threads to cores—a task that could be prohibitively expensive on a system with a large number of frequencies.

5.6 Summary

Scheduling for asymmetric multicore systems has been an active research area. It is generally agreed that asymmetry-aware thread schedulers are essential for achieving effective utilization of asymmetric systems. While early scheduling proposals suggested extensive sampling (running each thread on each core type) to determine good thread assignment, more recent research is converging on a combination of lightweight sampling (reading hardware performance counters on any core where the thread is running) and modeling (using that information to predict relative speedup). Most models developed so far were evaluated on systems where cores have identical microarchitecture but differ in frequency [17], [86] or retirement width [75]. What is missing, however, is a model that would work on systems where cores' microarchitecture differs more dramatically. For instance, Saez, Koufaty, and Ghiasi all relied on the amount of off-core or off-chip requests to model relative speedup on different core types. However, on their experimental systems all cores had the same cache size. It is not clear how well their models would work to predict relative speedup on a core whose cache size is different than that of the core where the rate of off-core requests was measured. After all, the rate of off-core requests may be strongly dependent on the cache size. We feel that developing models for asymmetric systems with more dramatic differences in cores' microarchitectures is a prominent gap that must be addressed by future researchers.

6 Conclusions and Future Directions

OS scheduler has been traditionally tasked with time-sharing cores among threads and balancing the load across computing units. In this work, we surveyed three exciting new areas where the OS scheduler must play a vital role. The scheduler was shown to be an essential part of energy-efficient and thermal-aware computing.

A common property of the surveyed algorithms is that they all relied on monitoring of what was going on in the hardware and in applications, and choosing the best resource allocation strategy based on that information. Although decisions themselves were in many cases straightforward, obtaining the right information was almost always challenging. For instance, the hardware does not provide a way to measure per-application power consumption, so researchers had to create intricate power models. Asymmetric processors do not reveal how much performance improvement an application is enjoying on the fast cores, so researchers had to approximate that information. Information flow is also weak on the part of applications. For instance, applications do not inform the system about their phase changes or about power- or thermal-intensity of particular regions of code. Knowing these details would greatly simplify the job of the OS scheduler.

We believe, therefore, that the future of research in this area is in effective codesign of the hardware, runtime management layer (including compilers), and applications. A solution that has all these components interacting in synergy to achieve system-wide efficiency goals will most certainly be better than solutions built into any single component in isolation.

Acknowledgments

This research was funded by the National Science and Engineering Research Council of Canada (NSERC) under the Strategic Project Grant program, by the Spanish government's research contracts TIN2012-32180 and the Ingenio 2010 Consolider ESP00C-07-20811 and by the $HIPEAC^2$ European Network of Excellence.

References

[1] T. Burd and R. Brodersen, "Energy Efficient CMOS Microprocessor Design," *Proc. 28th Hawaii Int'l Conf. System Sciences,* vol. 1, pp. 288-297, 1995.

[2] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS Digital Design," *IEEE J. Solid-State Circuits,* vol. 27, no. 4, pp. 473-484, Apr. 1992.

[3] S. Kaxiras and M. Martonosi, *Computer Architecture Techniques for Power-Efficiency.* Morgan & Claypool, 2008.

[4] M. Bohr, R. Chau, T. Ghani, and K. Mistry, "The High-k Solution," *IEEE Spectrum,* vol. 44, no. 10, pp. 29-35, Oct. 2007.

[5] T.J. Semiconductor, B. Doyle, M. Group, and I. Corporation, "Transistor Elements for 30 nm Physical Gate Lengths and Beyond," *Int'l Technology J.,* vol. 6, pp. 42-54, 2002.

[6] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar, "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-caches," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01),* pp. 147-157, 2001.

[7] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. Int'l Symp. Computer Architecture (ISCA '01),* pp. 240-251, 2001.

[8] N. Azizi, A. Moshovos, and F.N. Najm, "Low-Leakage Asymmetric-Cell SRAM," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED '02),* pp. 48-51, 2002.

[9] HP, Intel Microsoft, Phoenix and Toshiba, *Advanced Configuration and Power Interface (ACPI) 4.0 Specification,* 2010.

[10] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy," *Proc. First USENIX Conf. Operating Systems Design and Implementation (OSDI '94),* 1994.

[11] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy," *Proc. 36th Ann. Symp. Foundations of Computer Science (FOCS '95),* pp. 374-382, 1995.

[12] T. Ishihara and H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processors," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 197-202, 1998.

[13] H. Aydi, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems," *Proc. 22nd IEEE Real-Time Systems Symp. (RTSS '01),* pp. 95-105, 2001,

[14] D. Zhu, R. Melhem, and B. Childers, "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-processor Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 7, pp. 686-700, July 2003.

[15] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for A Given Power Budget," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 347-358. 2006.

[16] G. Dhiman and T.S. Rosing, "Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* pp. 207-212. 2007.

[17] J.C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A Comprehensive Scheduler for Asymmetric Multicore Processors," *Proc. Fifth European Conf. Computer Systems (EuroSys '10),* 2010.

[18] D. Shelepov, J.C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar, "HASS: A Scheduler for Heterogeneous Multicore Systems," *ACM SIGOPS Operating Systems Rev.,* vol. 43, no. 2, pp. 66-75, 2009.

[19] V.W. Freeh, D.K. Lowenthal, F. Pan, N. Kappiah, R. Springer, and B.L. Rountree, "Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 6, pp. 835-848, June 2007.

[20] G. Dhiman and T.S. Rosing, "System-Level Power Management Using Online Learning," *IEEE Trans. Computer-Aided Design Integrated Ciruits and Systems,* vol. 28, no. 5, pp. 676-689, May 2009.

[21] Y.C. Lee and A. Zomaya, "Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 8, pp. 1374-1381, Aug. 2011.

[22] G. Dhiman and T.S. Rosing, "Dynamic Power Management Using Machine Learning," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '06),* pp. 747-754, 2006.

[23] S. Herbert and D. Marculescu, "Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors," *Proc. Int'l Symp. Low power Electronics and Design (ISLPED '07),* pp. 38-43, 2007.

[24] Intel Corporation "Intel Core i7-900 Desktop Processor Extreme Ed. Series and Intel Core i7-900 Desktop Processor Series on 32-nm Process,"Specification, http://download.intel.com/design/processor/datashts/323252.pdf, 2010.

[25] Intel Labs, *The SCC Platform Overview,* http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf, May 2010.

[26] S. Cho and R.G. Melhem, "Corollaries to Amdahl's Law for Energy," *Computer Architecture Letters,* vol. 7, no. 1, pp. 25-28, Jan. 2008.

[27] S. Cho and R.G. Melhem, "On the Interplay of Parallelization, Program Performance, and Energy Consumption," *IEEE Trans. Parallel and Distributed Systems,* vol. 21, no. 3, pp. 342-353, Mar. 2010.

[28] E. Le Sueur and G. Heiser, "Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns," *Proc. Int'l Conf. Power Aware Computing and Systems (HotPower '10),* pp. 1-8, 2010.

[29] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," *Proc. 35th Ann. Int'l Symp. Computer Architecture (ISCA '08),* pp. 363-374, 2008.

[30] D. Shin and J. Kim, "Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* pp. 408-413, 2003.

[31] M. Curtis-Maury, J. Dzierwa, C.D. Antonopoulos, and D.S. Nikolopoulos, "Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS '06),* pp. 157-166, 2006.

[32] I. Kadayif, M. Kandemir, and I. Kolcu, "Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption in Chip Multiprocessors," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE '04),* pp. 1158-1163, 2004,

[33] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," *Proc. 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10),* pp. 129-142, 2010.

[34] M. Kondo, H. Sasaki, and H. Nakamura, "Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS," *ACM SIGARCH Computer Architecture News,* vol. 35, no. 1, pp. 31-38, 2007.

[35] N. Takagi, H. Sasaki, M. Kondo, and H. Nakamura, "Cooperative Shared Resource Access Control for Low-Power Chip Multiprocessors," *Proc. 14th ACM/IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '09),* pp. 177-182, 2009.

[36] R. Watanabe, M. Kondo, H. Nakamura, and T. Nanya, "Power Reduction of Chip Multi-Processors Using Shared Resource Control Cooperating with DVFS," *Proc. 25th Int'l Conf. Computer Design (ICCD),* pp. 615 -622, Oct. 2007.

[37] G. Dhiman, G. Marchetti, and T. Rosing, "Vgreen: A System for Energy Efficient Computing in Virtualized Environments," *Proc. ACM/IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '09),* pp. 243-248, 2009.

[38] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro,* vol. 28, no. 3, pp. 54-66, May 2008.

[39] M. Banikazemi, D. Poff, and B. Abali, "PAM: A Novel Performance/Power Aware Meta-Scheduler for Multi-Core Systems," *Proc. ACM/IEEE Conf. Supercomputing (SC '08),* pp. 1-12, 2008.

[40] R.L. McGregor, C.D. Antonopoulos, and D.S. Nikolopoulos, "Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors," *Proc. IEEE 19th Int'l Parallel and Distributed Processing Symp. (IPDPS '05),* p. 28a, 2005.

[41] A. Snavely, D.M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for A Simultaneous Multithreading Processor," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '02),* pp. 66-76, 2002.

[42] A. Merkel, J. Stoess, and F. Bellosa, "Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors," *Proc. European Conf. Computer Systems (EuroSys '10),* pp. 153-166, 2010.

[43] A. Fedorova, M. Seltzer, and M.D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '07),* pp. 25-38, 2007.

[44] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors," *Proc. Seventh Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '08),* pp. 220-229, 2008.

[45] K. Tian, Y. Jiang, and X. Shen, "A Study on Optimally Co-Scheduling Jobs of Different Lengths on Chip Multiprocessors," *Proc. Sixth ACM Conf. Computing Frontiers (CF '09),* pp. 41-50, 2009.

[46] C.J.M. Lasance, "Thermally Driven Reliability Issues in Microelectronic Systems: Status-Quo and Challenges," *Microelectronics Reliability,* vol. 43, no. 12, pp. 1969-1974, 2003.

[47] S. Zhang and K.S. Chatha, "Approximation Algorithm for the Temperature-Aware Scheduling Problem," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '07),* pp. 281-288, 2007.

[48] A.K. Coskun, T.S. Rosing, K.A. Whisnant, and K.C. Gross, "Temperature-Aware MPSoC Scheduling for Reducing Hot Spots and Gradients," *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC '08),* pp. 49-54, 2008.

[49] T. Chantem, R.P. Dick, and X.S. Hu, "Temperature-aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs," *Proc. Design, Automation and Test in Europe (DATE '08),* pp. 288-293, 2008.

[50] S. Sharifi, C. Liu, and T.S. Rosing, "Accurate Temperature Estimation for Efficient Thermal Management," *Proc. Ninth Int'l Symp. Quality Electronic Design (ISQED),* pp. 137-142, 2008.

[51] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-Aware Task Scheduling at the System Software Level," *Proc. ACM/IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '07),* pp. 213-218, 2007.

[52] D. Rajan and P.S. Yu, "Temperature-Aware Scheduling: When Is System-Throttling Good Enough?" *Proc. Ninth Int'l Conf. Web-Age Information Management (WAIM '08),* pp. 397-404, 2008.

[53] K. Stavrou and P. Trancoso, "Thermal-Aware Scheduling for Future Chip Multiprocessors," *EURASIP J. Embedded Systems,* vol. 2007, no. 1, pp. 40-40, 2007.

[54] A.K. Coskun, T.S. Rosing, and K. Whisnant, "Temperature Aware Task Scheduling in MPSoCs," *Proc. Design, Automation and Test in Europe (DATE '07),* pp. 1659-1664, 2007.

[55] M. Gomaa, M.D. Powell, and T.N. Vijaykumar, "Heat-and-Run: Leveraging SMT and CMP to Manage Power Density through the Operating System," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI),* pp. 260-270, 2004,

[56] X. Zhou, Y. Xu, Y. Du, Y. Zhang, and J. Yang, "Thermal Management for 3D Processors via Task Scheduling," *Proc. 37th Int'l Conf. Parallel Processing (ICPP '08),* pp. 115-122, 2008.

[57] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G.H. Loh, D. McCaule, P. Morrow, D.W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," *Proc. 39th IEEE/ACM Int'l Symp. Microarchitecture (MICRO-39),* pp. 469-479, 2006,

[58] M. Awasthi and R. Balasubramonian, "Exploring the Design Space for 3D Clustered Architectures," *Proc. Third IBM Watson Conf. Interaction between Architecture, Circuits, and Compilers,* Oct. 2006.

[59] I. Yeo, C.C. Liu, and E.J. Kim, "Predictive Dynamic Thermal Management for Multicore Systems," *Proc. 45th Ann. Design Automation Conf. (DAC '08),* pp. 734-739, 2008.

[60] A.K. Coskun, T.S. Rosing, and K.C. Gross, "Utilizing Predictors for Efficient Thermal Management in Multiprocessor SoCs," *IEEE Trans. Computer-Aided Design Integrated Circuits and Systems,* vol. 28, no. 10, pp. 1503-1516, Oct. 2009.

[61] A.K. Coskun, T.S. Rosing, and K.C. Gross, "Proactive Temperature Balancing for Low Cost Thermal Management in MPSoCs," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '08),* pp. 250-257, 2008.

[62] A.K. Coskun, T.S. Rosing, and K.C. Gross, "Proactive Temperature Management in MPSoCs," *Proc. 13th Int'l Symp. Low Power Electronics and Design (ISLPED '08),* pp. 165-170, 2008.

[63] R.Z. Ayoub and T.S. Rosing, "Predict and Act: Dynamic Thermal Management for Multi-Core Processors," *Proc. 14th ACM/IEEE Int'l Symp. Low Power Electronics and Design (ISLPED '09),* pp. 99-104, 2009.

[64] A. Kumar, L. Shang, L.-S. Peh, and N.K. Jha, "HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management," *Proc. 43rd Ann. Design Automation Conf. (DAC '06),* pp. 548-553, 2006.

[65] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin, "Dynamic Thermal Management through Task Scheduling," *Proc. IEEE Int'l Symp Performance Analysis of Systems and Software (ISPASS '08),* pp. 191-201, 2008.

[66] A.K. Coskun, T.S. Rosing, K.A. Whisnant, and K.C. Gross, "Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs," *IEEE Trans. Very Large Scale Integrated Systems,* vol. 16, no. 9, pp. 1127-1140, Sept. 2008.

[67] A.K. Coskun, T.S. Rosing, and K.C. Gross, "Temperature Management in Multiprocessor SoCs Using Online Learning," *Proc. 45th ACM/IEEE Ann. Design Automation Conf. (DAC '08),* pp. 890-893, 2008.

[68] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration," *Proc. 33rd Ann. Int'l Symp. Computer Architecture (ISCA '06),* pp. 78-88, 2006.

[69] R. Kumar et al., "Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO-36),* 2003.

[70] E. Humenay, D. Tarjan, and K. Skadron, "The Impact of Systematic Process Variations on Symmetrical Performance in Chip Multiprocessors," *Proc. Conf. Design, Automation and Test in Europe (DATE '07),* 2007.

[71] R. Kumar et al., "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA '04),* 2004.

[72] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures," *Proc. Third Conf. Computing Frontiers (CF '06),* 2006.

[73] A. Fedorova, J.C. Saez, D. Shelepov, and M. Prieto, "Maximizing Power Efficiency with Asymmetric Multicore Systems," *Comm. ACM,* vol. 52, no. 12, pp. 48-57, 2009.

[74] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, pp. 78-117, 1970.

[75] D. Koufaty, D. Reddy, and S. Hahn, "Bias Scheduling in Heterogeneous Multicore Architectures," *Proc. Fifth ACM European Conf. Computer Systems (EuroSys),* 2010.

[76] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer,* vol. 41, no. 7, pp. 33-38, 2008.

[77] J.C. Saez, A. Fedorova, M. Prieto, and H. Vegas, "Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors," *Proc. ACM Int'l Conf. Computing Frontiers (CF),* 2010.

[78] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's Law through EPI Throttling," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA '05),* pp. 298-309, 2005.

[79] J.C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar, "Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems," *IEEE Micro,* vol. 28, no. 3, pp. 26-41, May/June 2008.

[80] V. Kumar and A. Fedorova, "Towards Better Performance Per Watt in Virtual Environments on Asymmetric Single-ISA Multi-Core Systems," *ACM SIGOPS Operating Systems Rev.,* vol. 43, no. 3, pp. 105-109, 2009.

[81] T. Li et al., "Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures," *Proc. ACM/IEEE Conf. Supercomputing (SC '07),* 2007.

[82] A. Fedorova, D. Vengerov, and D. Doucette, "Operating System Scheduling on Heterogeneous Core Systems," *Proc. Workshop Op Sys Support for Heterogeneous Multicore Architectures,* 2007.

[83] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," *ACM SIGARCH Computer Architecture News,* vol. 33, no. 2, pp. 506-517, 2005.

[84] V. Kazempour, A. Kamali, and A. Fedorova, "AASH: An Asymmetry-Aware Scheduler for Hypervisors," *Proc. ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments (VEE),* 2010.

[85] J.A. Winter and D.H. Albonesi, "Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '08),* pp. 42-51, 2008.

[86] S. Ghiasi, T. Keller, and F. Rawson, "Scheduling for Heterogeneous Processors in Server Systems," *Proc. Second Conf. Computing Frontiers (CF '05),* pp. 199-210, 2005.

Sergey Zhuravlev received the MSc degree at Simon Fraser University in 2010 focusing his research on OS scheduling in multicore processors. Currently he is a design engineer at Teradici Corporation working on virtual desktop solutions.

Juan Carlos Saez received the PhD degree in computer science from the Complutense University of Madrid (UCM) in 2011 and the BA degree in music from Teresa Berganza Professional Music Conservatory in 2006. He is now an assistant professor in the Department of Computer Architecture at UCM. His research interests include energy-aware and cache-aware task scheduling on multicore and many-core processors. His recent research interests include operating system scheduling on heterogeneous multicore processors, exploring new techniques to deliver better performance per watt, and quality of service on these systems.

Sergey Blagodurov is currently working toward the PhD degree in School of Computing Science at Simon Fraser University, Vancouver, Canada. His research interests include resource contention-aware scheduling in high-performance computing (HPC) clusters of multicore machines and exploring new techniques to deliver better performance on nonuniform memory access (NUMA) multicore systems under Linux. He has also worked as a research associate at HP Labs, where he studied the ways to improve the efficiency of information collection and analysis in widely-used IT applications.

Alexandra Fedorova received the PhD degree from Harvard University in 2006, where she completed a thesis on operating system scheduling for multicore processors. She is an assistant professor of computer science at Simon Fraser University (SFU) in Vancouver, Canada. Concurrently with her doctorate studies, she worked at Sun Microsystems Research Labs, where she investigated transactional memory and operating systems. At SFU, she cofounded the SYNAR (Systems, Networking and Architecture) research lab. Her research interests include operating system and runtime design for multicore processors, with a specific focus on resource management.

Manuel Prieto received the PhD degree in computer science from the Complutense University of Madrid (UCM) in 2000. He is now associate professor in the Department of Computer Architecture at UCM and serves as the vice-dean for External Relations and Research at the Faculty of Informatics. His research interests include areas of parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and on complexity-effective microarchitecture design. His current research addresses emerging issues related to chip multiprocessors, with a special emphasis on the interaction between the system software and the underlying architecture. He has cowritten numerous articles in journals and for international conferences in the field of parallel computing and computer architecture. He is a member of the ACM.