Maximizing Heterogeneous Processor Performance Under Power Constraints

ALMUTAZ ADILEH, Ghent University
STIJN EYERMAN, Intel Belgium
AAMER JALEEL, Nvidia Research
LIEVEN EECKHOUT, Ghent University

Heterogeneous processors (e.g., ARM's big.LITTLE) improve performance in power-constrained environments by executing applications on the 'little' low-power core and move them to the 'big' high-performance core when there is available power budget. The total time spent on the big core depends on the rate at which the application dissipates the available power budget. When applications with different big-core power consumption characteristics concurrently execute on a heterogeneous processor, it is best to give a larger share of the power budget to applications that can run longer on the big core, and a smaller share to applications that run for a very short duration on the big core.

This article investigates mechanisms to manage the available power budget on power-constrained heterogeneous processors. We show that existing proposals that schedule applications onto a big core based on various performance metrics are not high performing, as these strategies do not optimize over an entire power period and are unaware of the applications' power/performance characteristics. We use linear programming to design the DPDP power management technique, which guarantees optimal performance on heterogeneous processors. We mathematically derive a metric (Delta Performance by Delta Power) that takes into account the power/performance characteristics of each running application and allows our power-management technique to decide how best to distribute the available power budget among the co-running applications at minimal overhead. Our evaluations with a 4-core heterogeneous processor consisting of big.LITTLE pairs show that DPDP improves performance by 16% on average and up to 40% compared to a strategy that globally and greedily optimizes the power budget. We also show that DPDP outperforms existing heterogeneous scheduling policies that use performance metrics to decide how best to schedule applications on the big core.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Scheduling**; **Power management;**

Additional Key Words and Phrases: Heterogeneous chip multiprocessors, scheduling, power management, DPDP

ACM Reference Format:
Almutaz Adileh, Stijn Eyerman, Aamer Jaleel, and Lieven Eeckhout. 2016. Maximizing heterogeneous processor performance under power constraints. ACM Trans. Archit. Code Optim. 13, 3, Article 29 (September 2016), 23 pages.
DOI: http://dx.doi.org/10.1145/2976739

This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295. This research was done when Stijn Eyerman was at Ghent University.
Authors' addresses: A. Adileh and L. Eeckhout, ELIS – Ghent University, iGent, Technologiepark 15, 9052 Zwijnaarde, Belgium; emails: almutaz.adileh@ugent.be, Lieven.Eeckhout@UGent.be; A. Jaleel, 392 Hudson St., Northborough, MA 01532; email: ajaleel@nvidia.com; S. Eyerman, Intel, Veldkant 31, 2550 Kontich, Belgium; email: Stijn.Eyerman@intel.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1544-3566/2016/09-ART29 $15.00
DOI: http://dx.doi.org/10.1145/2976739

1. INTRODUCTION

Technology scaling trends have forced processor designers into an era with new design constraints and challenges. Although transistors have become abundant, the active power consumption is expected to generate heat that far exceeds the ability to cool the processor. Consequently, the thermal characteristics of a processor have become a critical resource. In response, processor designers limit the total power consumption over a thermally significant time period by avoiding a large fraction of the processor from operating simultaneously, a phenomenon known as dark silicon [Esmaeilzadeh et al. 2011; Hardavellas et al. 2011]. Maximizing performance in the era of dark silicon requires novel techniques that optimally exploit the available power budget [Taylor 2012, 2013].

Optimizing processor performance under power constraints has been an important area of research. Dynamic Voltage and Frequency Scaling (DVFS) is a well-known mechanism for managing power, energy, and thermals in single-core and multicore processors. Several techniques [Cochran et al. 2011; Isci et al. 2006; Ma et al. 2011; Wang et al. 2009; Winter et al. 2010] have used DVFS to maximize performance while strictly maintaining processor power consumption under the allowed power cap. More recent proposals allow the processor the freedom of running at higher frequencies, thus exceeding the power budget, followed by stalling the processor to ensure that the average power consumption over the thermally significant period does not exceed the predefined limit [Raghavan et al. 2012, 2013b; Rotem et al. 2012].

While DVFS can improve performance under power constraints, transition latencies put a practical limit on how often a processor can change voltage and frequency settings over a given time interval. Furthermore, the supply-voltage range over which dynamic scaling can be performed has shrunk over the years, reducing the opportunity for DVFS. Consequently, both academia and industry have proposed Heterogeneous Chip Multiprocessors (HCMPs) [Kumar et al. 2003] to combat the limitations of DVFS. HCMPs (e.g., ARM's big.LITTLE) consist of high-performance 'big' cores and power-efficient 'little' cores. Recent commercial HCMP offerings include Samsung's Exynos 5 [Samsung Electronics 2013], NVIDIA's Tegra-3/Tegra-4 [NVIDIA 2011], and Intel's QuickIA [Chitlur et al. 2012].

The big cores of an HCMP are designed for maximum performance and tend to be power hungry, while the little cores are designed for maximum energy efficiency and have lower performance. The performance and power consumption of HCMPs is a function of the application to core mapping, with time spent on the big core being the determining factor. As a result, significant research work has focused efforts on a dynamic scheduler that selects the appropriate core type based on performance [Becchi and Crowley 2006; Koufaty et al. 2010; Lakshminarayana et al. 2009; Shelepov et al. 2009; Van Craeynest et al. 2012, 2013] or energy efficiency [Chen and John 2009; Ghiasi et al. 2005; Lukefahr et al. 2012]. Unfortunately, these proposals do not take into account processor power constraints.

This article focuses on HCMPs with a constrained power budget, that is, the processor cannot consume more than a fixed power budget over a specific time period (e.g., n Watt per m seconds) that is dictated by design parameters (e.g., thermal design specifications). Under such power budget constraints, applications can be executed on the big core only when sufficient power budget is available. Otherwise, the application must be executed on the little core[1]. Consequently, application performance on such systems directly depends on efficiently consuming the available power budget (which is a function of the application power consumption on the big core). Intuitively, the power budget should be distributed among concurrently executing applications based on utility, that

[1]In our setup, we assume that the power consumption of the little core never exceeds the power constraints, similar to the sustained-workload case in Jeff [2013].

is, the ability for an application to execute a large fraction of the defined time period on the big core. If an application can execute a large fraction of the power estimation period on the big core, it should be given a larger share of the power budget compared to an application that can run less on the big core and thus benefit less from running on the big core. With this in mind, this article makes the following contributions:

—To the best of our knowledge, we are the first to propose partitioning the power budget between concurrently executing applications on power-constrained HCMPs.
—We formulate the performance optimization problem on power-constrained HCMPs as a linear programming optimization. We show that the optimal solution is a schedule in which each application runs on either a big or a small core, and exactly one application runs partially on both.
—We show that, to obtain optimal performance on power-constrained HCMPs, big-core resources should be given to applications with the highest Delta Performance/Delta Power (DP/DP), that is, the ratio of the performance delta and the power delta between the big versus little core.
—We propose DPDP power budget partitioning, a novel policy that dynamically ranks and schedules applications to big and little cores based on the DP/DP metric. Our proposal uses the insight of the linear program solution to design a scalable power-budget partitioning policy, that is proven to be optimal in an offline scenario.
—A surprising (perhaps counterintuitive) finding is that memory-intensive applications tend to be preferred (over compute-intensive applications) to run on the big core in power-constrained environments. Because memory-intensive applications consume less power on the big core than compute-intensive applications, they can run a longer fraction of time on the big core before having to migrate to the little core. Therefore, in many cases, they better leverage the power budget to improve performance than compute-intensive applications.

Our evaluations with DPDP on a 4-core heterogeneous processor consisting of big.LITTLE pairs show that DPDP improves chip performance by 16% on average and up to 40% over a strategy that greedily and globally optimizes the power budget. We demonstrate that DPDP outperforms schedulers based on commonly used heuristics such as performance ratio and performance per Watt. We also show that DPDP is scalable to different core counts, core types, and power budgets. Moreover, we analyze the impact of DPDP on per-application performance, and we propose a technique to enforce a user-defined tolerable slowdown. Our results show DPDP's ability to maximize performance while maintaining the desired latency requirements.

2. MOTIVATION

2.1. Implications of Power Limits on HCMP Scheduling

We define a power constraint as the maximum power consumption averaged over a certain time interval, meaning that power consumption can temporarily exceed this limit, as long as it is followed by a lower power phase to ensure that the average is within the limit. This is different from prior work [Cochran et al. 2011; Isci et al. 2006; Ma et al. 2011; Wang et al. 2009; Winter et al. 2010], which typically assumes a strict power limit at every moment in time. This alternative definition follows the acceptable standard definition of thermal design power (TDP) for Intel and AMD processors [Huck 2011], for the sake of proper thermal management. Such a definition entails a maximum power value that can be drawn over a thermally significant time period. This power value can be exceeded instantaneously as long as it is followed by time periods in which the processor draws less than the allowed TDP to properly cool down the processor over the thermally significant period. Moreover, our adopted definition is also motivated by recent work on thermal management [Raghavan et al. 2012; Rotem et al. 2012]: heating

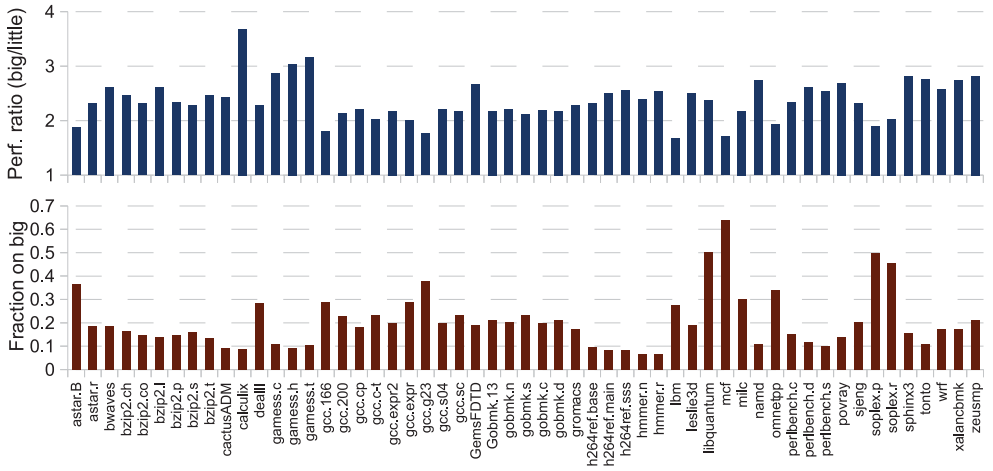


Fig. 1. The big/little performance ratio (top graph) and fraction of time that each application is allowed on the big core based on a 1W per second power budget (bottom graph).

because of high-power consumption happens gradually and has a certain delay (thermal time constant). As a result, chip temperature is determined by the average power consumption over this time period, rather than the instantaneous power consumption. Rotem et al. [2012] mention time periods of 30s to 60s. Alternatively, power supplies can also temporarily exceed their rated power number. Lefurgy et al. [2008] report that power supplies can overprovision during a 1s time period. We conservatively set the power averaging time period to 1s, but our technique can handle any time period setting (as long as it is long enough compared to the core migration time).

In our HCMP setup, this power constraint definition means that we can execute more programs on the big cores than the power budget allows, followed by a migration to the little cores to compensate for the overconsumption. Therefore, HCMP power management should consider both the performance and power characteristics of each program on each core type.

Assuming no power constraints, Figure 1 (top graph) shows the performance advantage for SPEC CPU 2006 applications when running on the big core relative to the little core. Throughout Section 2, we assume an out-of-order little core. In Section 6, we show results for both in-order and out-of-order little cores (see Section 5 for our experimental setup). Under no constraints, applications can observe anywhere from 2× to 4× better performance on a big core relative to a little core. However, on a power-constrained HCMP, the budget limits how long the application can execute on the big core. Once the power budget is depleted, the application must be executed on the little core. Assuming a power budget of 1W to spend over 1s per application, Figure 1 (bottom graph) illustrates the fraction of the total execution time that each application can execute on the big core. Under power constraints, we observe that applications can spend as little as 10% of the total execution time on the big core (e.g., hmmer), or as much as 60% of the total execution time on the big core (e.g., mcf). The varying behavior among workloads is primarily due to the difference in power consumption on the big core. In general, we find that memory-intensive applications tend to have lower power consumption on the big core since they spend a large fraction of the execution time stalled waiting for memory, which enables them to spend more time on the big core for a given budget.

2.2. Power-Budget Partitioning

Based on the observations from the previous section, we now show how prior proposals are unsuitable for power-limited HCMP environments. Figure 2 shows an example

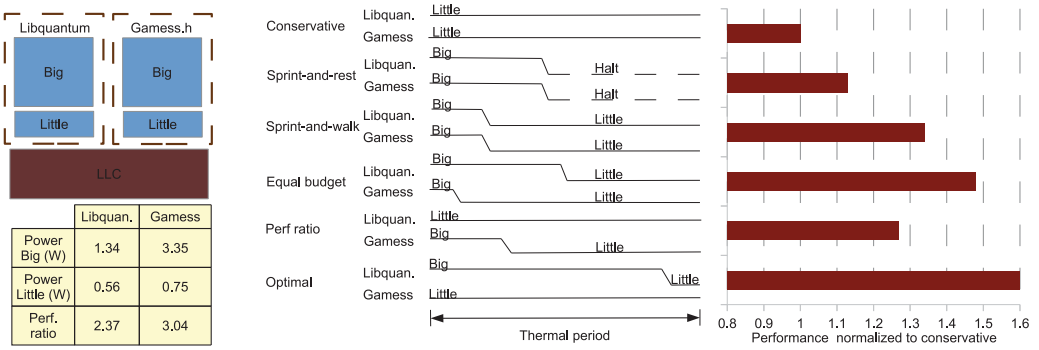


Fig. 2. Performance gain for several budget-partitioning approaches normalized to running all applications on the little cores.

heterogeneous multicore featuring two big and two little cores, concurrently running two applications: gamess.h and libquantum. The power consumption of both applications on each core type is provided as well. For this example, we assume a power budget of 2W over a period of 1s (1W per big.LITTLE pair).

The gamess.h benchmark is a compute-intensive workload that significantly benefits from the big core (3× performance), but if it may only consume 1W, it can be run on the big core for 0.09s only. For the remaining 0.91s, it has to run on the little core because of its relatively high-power consumption on the big core. Although the memory-intensive libquantum does not benefit from the big core as much (2.3× performance), its relatively low-power consumption allows it 0.5s on the big core for the same 1W power consumption. In total, libquantum achieves about 40% higher performance than gamess.h (both relative to little core) when both are given the same 1W per second power budget.

Figure 2 also shows the performance (drawn to scale) for several HCMP scheduling approaches. While not all of these approaches explicitly partition the power budget, the application to core mapping indirectly partitions the power budget based on which and when applications execute on a big core:

—The *conservative approach* interprets the power budget as a strict power limit (total power cannot exceed 2W at any time). If the total power consumption of executing one or more applications on the big core exceeds the power budget (as is the case in our example), the applications can execute only on the little core. Consequently, this approach does not utilize the available power budget, thus has suboptimal performance. This approach is taken by most DVFS-based CMP power-capping studies [Cochran et al. 2011; Isci et al. 2006; Ma et al. 2011; Wang et al. 2009; Winter et al. 2010].
—*Sprint-and-rest* is similar to computational sprinting for long-running applications [Raghavan et al. 2012, 2013b]. Here, we execute all applications on the big core to obtain the highest performance; as soon as we have consumed the available budget, the HCMP is turned off to cool down.
—*Sprint-and-walk* uses a similar approach to sprint-and-rest, but after sprinting both applications on the big core, we move both of them to the little cores such that the total budget is still preserved. It is clear that the fraction spent on the big core for both applications will shrink compared to sprint-and-rest to provision for the run to continue on the little cores. This is the HCMP scheduling variant of Intel's Turboboost 2.0 [Rotem et al. 2012], which increases the frequencies of all cores if there is thermal headroom.

—*Equal budget partitioning* divides the power budget equally among the applications (each getting 1W per second). Here, each application spends a different fraction of the time on the big core based on its power rates on the big and little cores.
—*Performance ratio* ranks the applications by their big-to-little performance ratio. We always run the lowest ranked application on the little core while the highest ranked application gets the remainder of the budget (which allows it to run a fraction of the time on the big core). This is a common approach for scheduling in HCMPs.
—**Optimal** system performance is achieved by favoring libquantum over gamess.h, that is, run gamess always on little, and give the remaining budget to libquantum to run on the big core.

The suboptimal performance observed for the various scheduling policies is mainly due to being application-unaware. Both sprint-and-rest and sprint-and-walk let all the applications greedily compete for the budget: the applications with higher power-consumption rates deplete most of the budget, leaving the lower-power applications with a smaller fraction of the budget despite being better at utilizing it. Similarly, although the performance-ratio approach tries to optimize where to allocate its budget, ignoring the power limits restricts the time spent on the big core, leading to a wrong prediction of which application would benefit the most from the given budget. Although equal budget partitioning provides an equal chance for both applications, it fails to reach the optimal performance because the budget given to gamess.h is depleted quickly, not benefiting its total performance significantly. However, when prioritizing libquantum, its memory-intensive nature leads to lower power consumption that results in an overall higher utilization of the big core, which leads to higher overall system performance. The bottom line is that application awareness is essential to partition the available power budget among co-running applications to maximize overall system performance.

Maximizing performance in power-constrained HCMPs mandates optimally tuning the fraction of time that each application gets on the big core, which comes down to searching through an infinite number of possible fraction allocations. This analysis clearly motivates the need for a new optimal and scalable mechanism for partitioning the available power budget across concurrently executing applications. To that end, the next section formulates the power-budget partitioning problem using linear programming, which yields a practical, yet well-performing algorithm.

3. POWER-BUDGET PARTITIONING USING LINEAR PROGRAMMING

As shown in the previous section, partitioning the power budget across applications to optimize performance is not straightforward. A partitioning policy should take into account both the performance gain of an application on the big core and the fraction of time that it can spend on the big core, which is determined by its power consumption. Instead of trying out various heuristics, we take a more rigorous approach by formulating the problem statement using linear programming. Note that the power manager itself does not need to solve a linear program during runtime. Instead, the key insight from the mathematical formulation leads to a solution that enables a low-overhead scalable power manager to dynamically find the optimal schedule and power distribution among the applications in the large design space.

3.1. Linear Programming Formulation

To formulate power-budget partitioning as a linear programming problem, we denote performance as S and power consumption as P (in Watts). The performance of each application is expressed as its instructions per second (IPS) divided by its IPS when run on the big core in isolation (i.e., its weighted IPS), such that the sum of the

performance of all applications in the workload equals system throughput (STP) [Eyerman and Eeckhout 2008]. $S_{L,i}$ and $P_{L,i}$ denote performance and power, respectively, for application i on the little core, whereas $S_{B,i}$ and $P_{B,i}$ denote performance and power on the big core. f_i denotes the fraction of the power averaging time period application i executes on the big core; by consequence, $1 - f_i$ then is the fraction of time that it runs on the little core. P_{budget} is the power budget. Our objective is to find f_i for each application i, so that STP is maximized while remaining within the power budget. We only consider solutions in which each application either runs on the big or the little core (no idle periods), because we find a sprint-and-rest scheme to be always suboptimal for our configuration. This optimization problem can be written as a linear programming problem, as shown in Equation (1):

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{n} f_i S_{B,i} + (1 - f_i) S_{L,i} \\ \text{subject to } & 0 \leq f_i \leq 1, \forall i \\ & \sum_{i=1}^{n} f_i P_{B,i} + (1 - f_i) P_{L,i} \leq P_{budget} \end{aligned} \quad . \tag{1}$$

It is clear that the set of fractions f_i that meet the constraints to form a correct solution is infinite. However, an interesting characteristic of linear programming is that an optimal solution is at one of the intersection points of the constraint equations. In the case of n applications, however, finding a solution could be cumbersome because a comprehensive search to find and evaluate the intersection points is still needed. Nevertheless, we will show how we circumvent this obstacle by exploiting an important characteristic of the solution space, as we describe next.

3.2. The Solution Space

To ease the discussion, we first consider two applications, then generalize our findings to more applications. For two applications, the problem can be rewritten as

$$\begin{aligned} \text{maximize } & f_1 S_{B,1} + (1 - f_1) S_{L,1} + f_2 S_{B,2} + (1 - f_2) S_{L,2} \\ \text{subject to } & 0 \leq f_1, f_2 \leq 1 \\ & f_1 P_{B,1} + (1 - f_1) P_{L,1} + f_2 P_{B,2} + (1 - f_2) P_{L,2} \\ & \quad \leq P_{budget} \end{aligned} \quad . \tag{2}$$

The solution space of this optimization problem is shown in Figure 3, left-hand side. f_1 and f_2 need to be inside the square between 0 and 1, and the power budget restricts the solutions to the left of the line cutting the square. Due to the nature of linear programs, the optimal solution is one of the two intersections of the budget line and the square (indicated by the dots). This means that there are only two possibly optimal solutions: either program 1 *or* program 2 runs on the big core as long as possible, and if any budget is left over, the other program can run on the big core for a fraction of the time only.

A similar argument can be made for multiple applications in n dimensions: the optimal solution is always on one of the edges of the unit hypercube, meaning that only one fraction is a real number between 0 and 1, and all other fractions are either 0 or 1. To illustrate this, the right part of Figure 3 shows six possible solutions in three dimensions: all solutions have two fractions either 0 or 1, and one fraction in between 0 and 1. This implies that all applications run either on the big core or the little core all of the time, and *one (and only one!)* application switches between big and little (because its fraction is in between 0 and 1). Finding a solution thus boils down to *finding which*

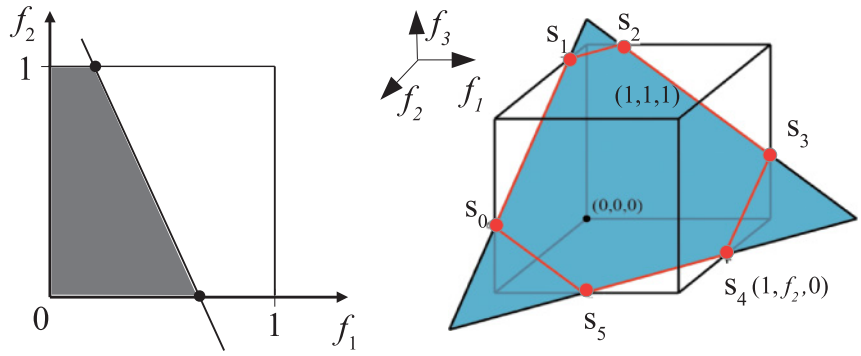


Fig. 3. Graphical representation of the solution space for two-program (left) and three-program (right) combinations. The diagonal line/plane represents the power budget. The shaded area indicates the solution space, and the dots are potential optimal solutions.

applications to always run on the big core (if any), which applications to always run on the little core, and finding the one application that should switch between core types.

3.3. Delta Performance/Delta Power

We have shown that, using linear programming optimization, an infinite solution space can be reduced to prioritizing which applications to run on the big core at the availability of a power budget. However, searching comprehensively through all possible solutions is still not a feasible approach for a dynamic power manager.

The question now is how to rank the applications such that the top-ranked applications run on the big core and the bottom-ranked applications run on the small core; the application at the boundary then needs to switch between the big and small cores. To derive a mathematically sound ranking metric, we analytically solve the linear program. We first do the analysis for two applications only, then generalize our findings to more applications.

Using the problem defined in Equation (2) for two applications, we note that the optimum is achieved when the budget is completely consumed, making the second restriction an equation instead of an inequality. We solve this equation for f_2, and replace f_2 in the maximization function with that expression. This yields a linear function in f_1:

$$\begin{aligned} &\text{maximize } \alpha f_1 + \beta, \\ &\text{with } \alpha = \frac{S_{B,1} - S_{L,1}}{P_{B,1} - P_{L,1}} - \frac{S_{B,2} - S_{L,2}}{P_{B,2} - P_{L,2}}. \end{aligned} \tag{3}$$

Maximizing this function depends on the sign of α: if α is positive, f_1 should be as large as possible; if α is negative, f_1 should be as small as possible. The sign of α is determined by the DP/DP ratio: if the difference in performance between the big and little core divided by the difference in power consumption between the big and little core for program 1 is larger than for program 2, the sign is positive, and vice versa. Thus, if the DP/DP ratio of program 1 is larger than for program 2, program 1 should execute on the big core as long as possible; if it is smaller, program 2 should run on the big core.

Applying the same solution method for three programs yields the following result (with $DPDP_i$ the DP/DP ratio of application i between big and little core, and β a constant term):

$$\text{maximize } (DPDP_1 - DPDP_3)\, f_1 + (DPDP_2 - DPDP_3)\, f_2 + \beta. \tag{4}$$

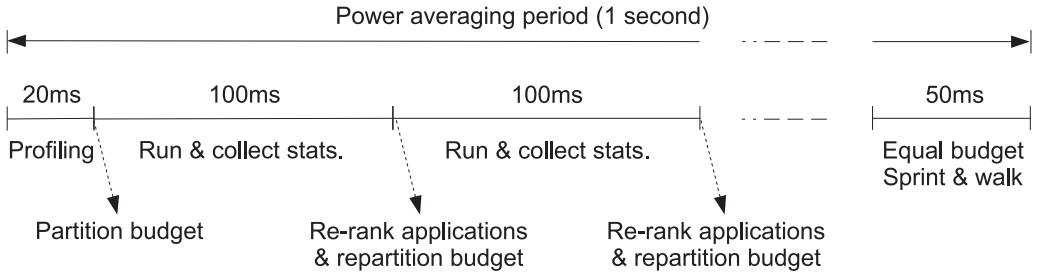


Fig. 4. The four phases of the DPDP power manager.

This means that if $DPDP_1$ is larger than $DPDP_3$, f_1 should be maximized, and similarly for f_2. If $DPDP_1$ is smaller than $DPDP_3$, then f_1 should be minimal, and similarly for f_2. If both $DPDP_1$ and $DPDP_2$ are larger than $DPDP_3$, then the largest of $DPDP_1$ and $DPDP_2$ will determine which fraction yields the largest performance benefit: if $DPDP_1$ is larger than $DPDP_2$, the term with f_1 will be larger than the term with f_2; thus, maximizing f_1 yields the largest performance benefit, and vice versa for f_2. In conclusion, the ideal scheduling policy is to select the program with the largest DPDP to run on the big core, and if budget is left, select the second largest DPDP, and so on. A similar analysis for four programs gives the same conclusion (not shown due to space constraints).

The insights gained from the linear program analysis provides us with the foundation for an optimal schedule: rank the programs based on DP/DP, and calculate the fraction that the highest-ranked program can run on the big core, assuming all other programs execute on the little cores. If that fraction is smaller than 1, the optimal schedule is found. If it is 1, calculate how long the program ranked second can execute on the big core, given that the first program runs on the big core all of the time and the other programs execute on the little core. Continue this process until the budget is fully consumed. This is a linear method in the number of programs, which makes it a scalable solution.

4. DPDP BUDGET PARTITIONING

The mathematically derived optimal power management foundations described in the previous section assume that performance and power consumption are known for all applications for both the big and little cores. Moreover, it is assumed to be constant over the thermally significant period. In reality, this is not the case: performance and power are unknown (or need to be measured or predicted across core types), and applications go through phase changes during execution. In this section, we discuss the implementation details of our power manager, called DPDP, which leverages the key insights described in the previous section to optimize performance within a tight power budget in a low-overhead and scalable way.

DPDP requires hardware support to independently operate (and deactivate) individual cores in the processor in addition to the ability to measure the performance and power consumption of each core in the processor as the applications run.

DPDP power-budget partitioning involves four phases: (i) profiling, (ii) a ranking and partitioning phase, (iii) a monitoring and repartitioning phase to adapt to application phase changes, and (iv) sprint-and-walk to make up for profiling inaccuracies and to ensure that we do not exceed the power budget. Figure 4 shows how these phases are distributed along the thermally significant time period.

Phase #1: Initial profiling. This phase is done only once, when the applications start. The profiling is done by executing each application for a short duration on each

core type and measuring its performance and power consumption. To set the duration of the profiling phase, we need to make a compromise between profiling accuracy and overhead. A longer profiling phase has a better chance of capturing accurate power and performance measurements for each application. However, it allows applications to inefficiently consume part of the power budget, reducing the potential performance gain. We set our profiling duration to 10ms on each core type for a total overhead of 2% for a power-averaging period of 1s (2 times 10ms). For applications that have no fine-grained phase behavior, this duration could be reduced without losing accuracy. We profile all co-running applications in parallel to reduce the overhead and to capture the effect of interference in shared resources. We start by running half of them on the big cores and the other half on the little cores, and switch after 10ms.

Phase #2: Ranking applications and partitioning the budget. As discussed in Section 3, the optimal schedule requires the applications to run either on the big core or the little core, except for one application that runs partially on both core types. Using the statistics gathered for each application in the profiling phase, our scheme ranks the applications based on their respective DP/DP metrics, and uses this ranking to determine the schedule for each application. Algorithm 1 summarizes the classification and partitioning phase. The algorithm starts with the highest-ranked application and assumes that all the other applications run on the little cores. If the remaining budget permits, the scheduler allocates a big core to this application and allocates the required power budget for that core. Then, it updates the remaining budget statistics. The scheduler repeats the same procedure iteratively for the remaining applications in rank order. Once an application cannot fully execute on the big core, the scheduler calculates the fraction of time that the application is permitted to run on the big core, and schedules the remaining applications on the little cores.

ALGORITHM 1: Determining the Fraction of Time on the Big Core that Each Application Gets During a Power-Averaging Period

Start with list of applications ranked by DP/DP
$consumed\_budget = \sum$(power of all apps on little core)
while $consumed\_budget < available\_budget$ **do**
  Take the next highest-ranked application a
  if $available\_budget - consumed\_budget \geq P_{B,a} - P_{L,a}$ **then**
    Schedule application a on big all time
    $consumed\_budget = consumed\_budget - P_{L,a} + P_{B,a}$
  else
    $Fraction_{big}(a) = \frac{available\_budget - consumed\_budget}{P_{B,a} - P_{L,a}}$
    Budget fully consumed, end while loop
  end if
end while
Schedule the rest of the applications on little core

Phase #3: Statistics collection and budget repartitioning. To cope with changes in the application-phase behavior, our scheme continuously accumulates power and performance statistics for each application based on its allocated core type. Every 100ms, our scheme repeats Phase #2 using the updated performance and power values in addition to the total power consumed up to this point. This enhances the accuracy of the measured statistics and ensures the adaptability of our power-budget partitioning scheme to changes in workload behavior.

Phase #4: Sprint-and-walk at the end of the power period. In the last 50ms, we determine the leftover budget. We equally divide this budget among the applications,

Table I. Big and Little Core Configurations

	Big	Little
Type	Out-of-order	In-order
Frequency	2.6GHz	1.5GHz
Voltage	0.9V	0.64V
Pipeline width	4	2
ROB size	168	-
L1 I-cache	32KB	32KB
L1 D-cache	32KB	32KB
Shared L2 cache	4MB per pair	
Memory bandwidth	25.6GB/s	

and execute all of them on the little cores for 10ms. We then determine how much power is 'saved' by running on the little core compared to the allocated budget. We then 'burn' this excess power by running the applications on the big cores, until it is completely burned. After that we again execute on the little core, saving budget, then burn the saved power on the big core. This is repeated until the end of the power-averaging period. We call this *dynamic sprint-and-walk*: the fraction of time to run on the big core is dynamically determined by saving and burning the power budget. This step is required for two reasons. The first is to ensure that the execution remains within the power limit at the end of the period. The second reason is that we can use this phase as the profiling phase for the next power-averaging period. During Phase #3, most of the applications run on a single-core type for the whole duration. In Phase #4, on the other hand, each application runs on both the little and big core for some time, generating profile information for the next power-averaging period.

The overhead of the scheduler is minimal. The main overhead incurred by the scheduler is to rank the n applications, which has a complexity of $O(n \log n)$. Considering that this overhead is incurred at most once per 100ms (which is an adjustable design knob), the scheduler has an unnoticeable impact on performance. Moreover, by continuously monitoring an application's power and performance statistics in Phase #4, as described earlier, profiling overhead is incurred only at the beginning of the application run.

5. EXPERIMENTAL SETUP

We use the Sniper 6.0 [Carlson et al. 2014] simulation infrastructure (using its most detailed cycle-level core modes) to carry out the experiments in this article. We simulate heterogeneous multicore systems that consist of two core types, big and little (see Table I). The big core is an aggressive four-wide out-of-order core running at 2.6GHz, while the little core is a two-wide in-order core running at 1.5GHz. The last-level cache is shared by all cores. There is 4MB of LLC per pair of big and little cores.

We use the in-order little core configuration throughout Section 6. We consider a two-wide out-of-order little core in only one of the sensitivity studies to resemble recent low-power microarchitectures, such as Intel's Silvermont [Kuttana 2013]. We evaluate scheduling 4 applications on processors consisting of 4 pairs of big and little cores. We also demonstrate the applicability of our method to architectures having fewer big cores than little cores.

We use McPAT 1.3 [Li et al. 2009] to estimate the power consumption of our schedules, assuming a 22nm chip technology. We report total power consumption as the sum of the leakage power and the runtime dynamic power, assuming clock gating for unused structures in the active cores. Idle cores are power gated. We set the power budget for each big–little pair at 1W for each period of 1s, that is, 4 pairs of big and little cores are given 4W every second. This budget assumption is reasonable for the sake of our analysis, as it falls between the big core and little core power ratings and allows

sufficient room for optimization. A similar power budget has been assumed in prior work [Raghavan et al. 2012]. Moreover, we provide a sensitivity study to show the benefit of DPDP, as we vary the assumed baseline power budget. Our simulation infrastructure accounts for the overheads associated with migrating applications between cores. This includes 20 μs required for saving and restoring architectural state [Greenhalgh 2011] and for powering on the other core (because our scheduler knows when to migrate, powering on the other core could also be done slightly before the transition time). We also model the impact of cache warmup (on top of the 20 μs mentioned earlier). Overall, our power manager suffers minimal overhead because it switches between cores at most once every 100 ms in phase #3, and less than five times in phase #4.

To evaluate our scheme, we use all 26 SPEC CPU2006 benchmarks and consider all of their reference inputs, resulting in 55 benchmark–input combinations. We use PinPoint [Patil et al. 2004] to generate representative regions of 10 billion instructions, and we simulate 1s of execution. We consider 75 randomly chosen combinations of 4 benchmarks. We evaluate performance using total STP, which reflects the overall achieved throughput of the system compared to a reference single big core. We also consider user-perceived performance by evaluating the average normalized turnaround time (ANTT) [Eyerman and Eeckhout 2008].

6. RESULTS AND DISCUSSION

We now demonstrate the effectiveness of DPDP power-budget partitioning. We consider the following five schemes and evaluate their effectiveness at improving performance within the power budget of 1W per second per application.

—*Global sprint-and-walk.* Our first scheduler considers a global power budget (i.e., 4W per second for four applications), and greedily optimizes performance within the given power budget. It starts by executing all applications on the little cores for 10ms. It then calculates the 'saved' budget compared to the total budget, which it then burns by executing all applications on the big cores. The saved budget equals the available budget (0.01J per 10ms per application) minus the amount of energy consumed during the 10ms time interval. Once the available budget is burned, all applications migrate back to the little cores, saving the budget again for the next 10ms, which can then be burned on the big cores, and so on.

—*Equal budget sprint-and-walk.* This scheduler is similar to the previous one, except that we now partition the overall power budget across the co-running applications, and optimize the power budget for each application individually, that is, we assign 1W per second for each application. Similar to the previous scheduler, all applications start running on the little cores for 10ms. For each application, we calculate the saved budget relative to the available budget, and we greedily run the application on the big core until the saved budget is consumed. Once an application's power budget is consumed, it migrates back to the little core for another 10ms to build up its power budget again, and the scheme repeats.

—*Budget partitioning using performance ratio.* This scheduler is similar to DPDP as described in Section 4, but instead of using DP/DP as the ranking metric, we use performance ratio between big and little cores. In other words, applications that speed up more on the big core are given a larger share of the budget and thus higher priority to run on the big core as long as the power budget is not exceeded.

—*Budget partitioning using performance per Watt.* Here, we rank the applications based on the performance per Watt on the big core. Performance per Watt is a commonly used metric for expressing power efficiency; intuitively, it makes sense to run applications with the highest performance per Watt ratio on the big cores.

—*Budget partitioning using DP/DP.* This is the DPDP scheduler, as described in Section 4.

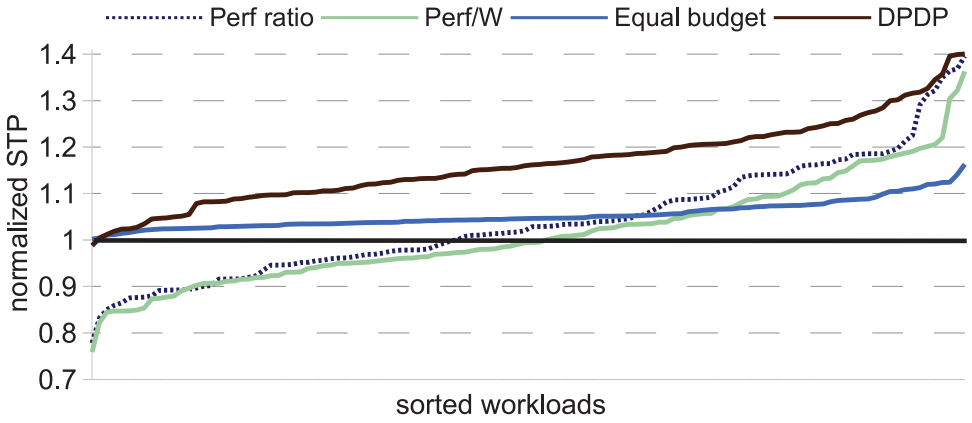


Fig. 5. Comparing the various power-budget partitioning schemes relative to global sprint-and-walk for mixes of four applications.

We normalize all of the results to the global sprint-and-walk scheme because this scheme is the natural translation of Intel's TurboBoost [Rotem et al. 2012], originally designed for DVFS, to HCMPs. The graphs in this section show how each of the schemes perform compared to the baseline scheme using an S-curve, showing the sorted relative performance difference for all workload combinations.

6.1. DPDP Results

Figure 5 quantifies the performance improvements achieved by DPDP for mixes of four applications. The graph clearly shows that DPDP outperforms the other power-budget partitioning schemes. DPDP improves performance by 16% on average and up to 40% over global sprint-and-walk for mixes of four applications. The performance improvement of DPDP stems from optimal budget partitioning. DPDP selects the applications that achieve the highest raise in performance given the available budget, the period over which power is calculated, and the performance characteristics of the application on both core types. The other alternatives, as explained in Section 2, fail to consider one or more aspects of performance maximization under a power limit.

The figure also demonstrates DPDP's robustness: DPDP improves overall performance for all workload mixes. Although equal budget partitioning consistently improves performance, for most mixes, the improvement is limited to less than 5% on average. The other two budget partitioning schemes are less robust, and do not consistently improve performance. In fact, about half of application mixes observe a performance degradation for the schemes based on the *performance ratio* and *performance per Watt* metrics. This clearly demonstrates the effectiveness of the DP/DP metric for application scheduling and power-budget partitioning.

Figure 6 shows the average performance improvement for DPDP over global sprint-and-walk for different mixes of compute-intensive and memory-intensive applications. We classify applications as memory-intensive if they spend at least 25% of their execution time waiting for main memory. We consider workload mixes with zero to up to four memory- and compute-intensive applications. The performance gain for DPDP over global sprint-and-walk peaks for mixes with 2 compute- and 2 memory-intensive applications. This is as expected: the larger the difference is between the applications' big-versus-little characteristics, the larger the impact of power-budget partitioning is on performance.

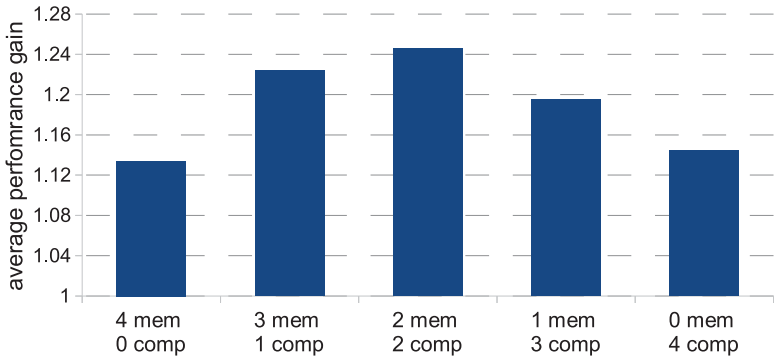


Fig. 6. Average performance improvement for DPDP versus global sprint-and-walk for different classes of compute- and memory-intensive four-application mixes.

6.2. Big-Core Utilization

To gain more insight into the performance benefits achieved through DPDP, we now investigate which applications get to run on the big core more frequently. Figure 7 breaks down the time spent on the big cores by application type (memory-intensive vs. compute-intensive), for DPDP versus budget partitioning using performance ratio. For a mix of four applications, the highest utilization of the available 4 big cores is 4. All the mixes shown in the figure use two memory-intensive and two compute-intensive applications.

Two observations can be made from the figure. First, DPDP leads to a higher big-core utilization compared to budget partitioning using the performance ratio metric (compare Figure 7(a) versus (b)). This suggests that DPDP is better able at effectively utilizing big-core resources, which explains the observed performance benefits.

Second, and more interestingly, DPDP tends to favor memory-intensive applications by allocating a larger fraction of the power budget to them than to compute-intensive applications, although not uniformly so—it is a function of the DP/DP ratio. This observation suggests that memory-intensive applications are better at utilizing the available budget than their compute-intensive counterparts. This is counterintuitive, as memory-intensive applications usually show a smaller performance benefit from running on a big core compared to compute-intensive applications. Infact, Becchi and Crowley [2006], Chen and John [2009], Ghiasi et al. [2005], Koufaty et al. [2010], and Shelepov et al. [2009] propose scheduling compute-intensive applications on a big core to optimize performance (in the absence of a power limit). Van Craeynest et al. [2012] show that memory-intensive applications could benefit from running on a big core by exploiting more memory-level parallelism, which explains the fact that the performance ratio metric also selects the memory-intensive applications for some mixes. However, we find that memory-intensive applications have another benefit under power constraints. Due to the fact that they wait more for main memory, they can more extensively leverage clock gating, which reduces the big core's power consumption. This, in turn, increases the time that they can spend on the big core, which leads to an overall increase in STP.

6.3. Sensitivity Analysis

We now explore the sensitivity of DPDP with respect to the available power budget, the core types available in the HCMP, and asymmetry in the HCMP configuration.

6.3.1. Available Power Budget. The available power budget has a considerable impact on the performance gain that can be achieved through power budget partitioning.

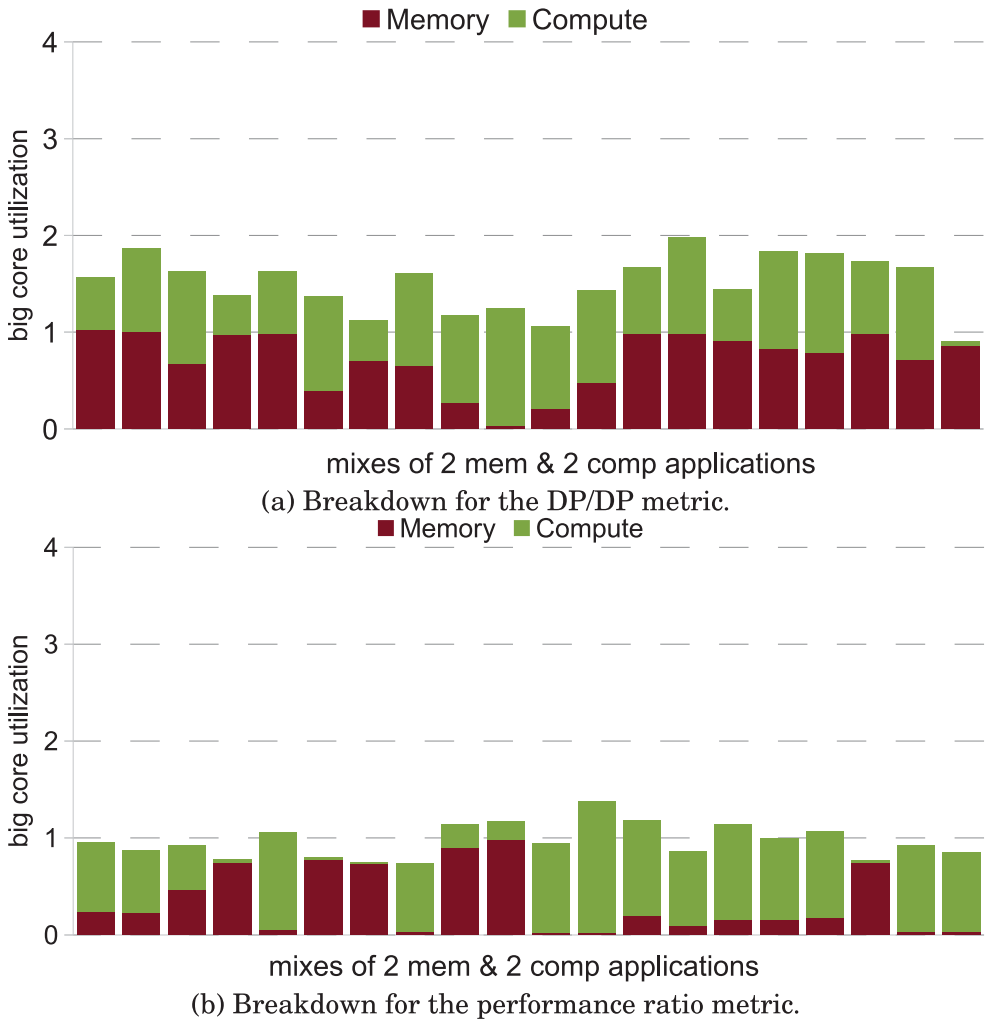


Fig. 7. Big-core usage (out of 4 cores). For most cases, DP/DP favors memory-intensive applications, achieving 55% higher big-core utilization than performance ratio.

Figure 8 shows the impact of varying the power budget on the achieved gain. Slightly decreasing the budget to 0.75W per second slightly decreases the average performance gain to 13.5%. Similarly, slightly increasing the budget to 1.5W per second shows smaller gains compared to the nominal 1W per second power budget. A much larger budget (2W per second), on the other hand, shows an insignificant performance gain. This is to be expected: for a power budget in between the power ratings of the big and little cores, proper power budget partitioning is expected to provide significant performance gains. Once the budget becomes either too constrained or too abundant relative to the little and big core's power consumption, budget partitioning becomes less valuable. For constrained cases, most of the applications would have to run on the little cores anyway, making it close to a conservative approach. For abundant budgets, on the other hand, most of the applications would be able to run on the big cores, limiting the opportunity for budget partitioning.

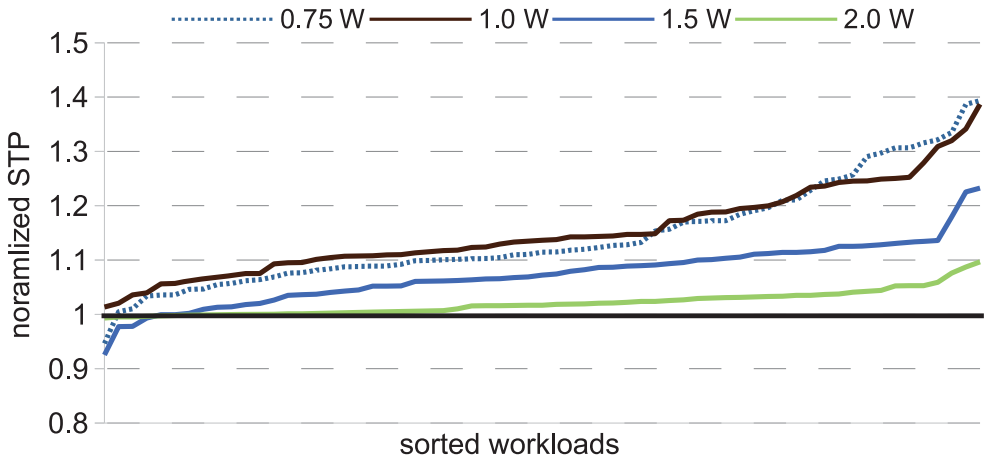


Fig. 8. Normalized STP across different power budgets.

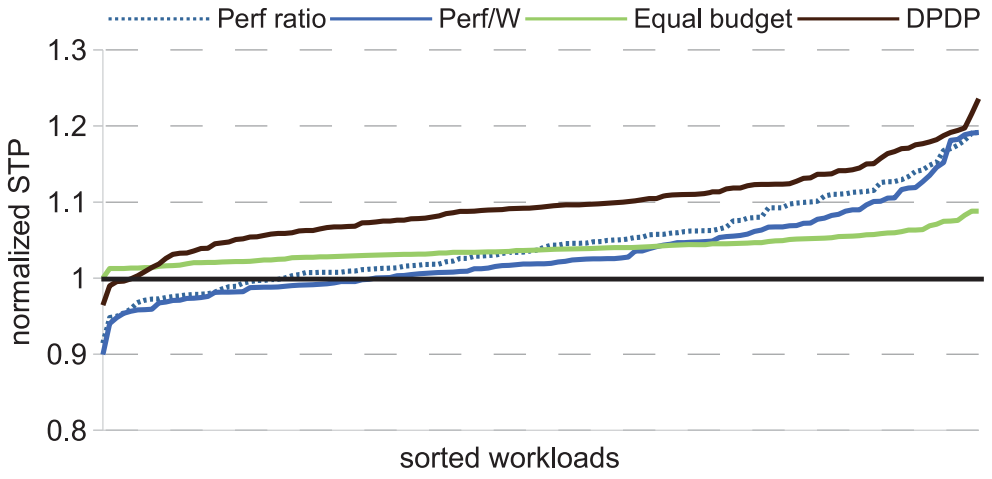


Fig. 9. Normalized STP assuming out-of-order little cores.

6.3.2. Core Type. We now set the little core to be an out-of-order core instead of an in-order core (frequency settings, cache hierarchy, and other structures remain the same; see Figure 9). DPDP still provides significant performance gains compared to a global sprint-and-walk approach. The improvement seen for a configuration with an out-of-order little core reaches a significant 9% on average and up to 26%. Note that the performance gain of an out-of-order little core is lower than the gain seen for the in-order configuration. This relatively lower performance gain happens for two reasons. First, the less powerful in-order little core provides relatively lower performance compared to the out-of-order little core, increasing the difference between the optimal and a suboptimal partitioning. Second, the in-order little core consumes less power than the out-of-order little core, which increases the fraction of time allowed on a big core for our budget partitioning scheme.

6.3.3. Asymmetric CMP Configuration. In the previous results, we assume as many big and little cores as there are applications. However, the DPDP scheduler also applies to configurations with fewer big cores than little cores. The only change is that the

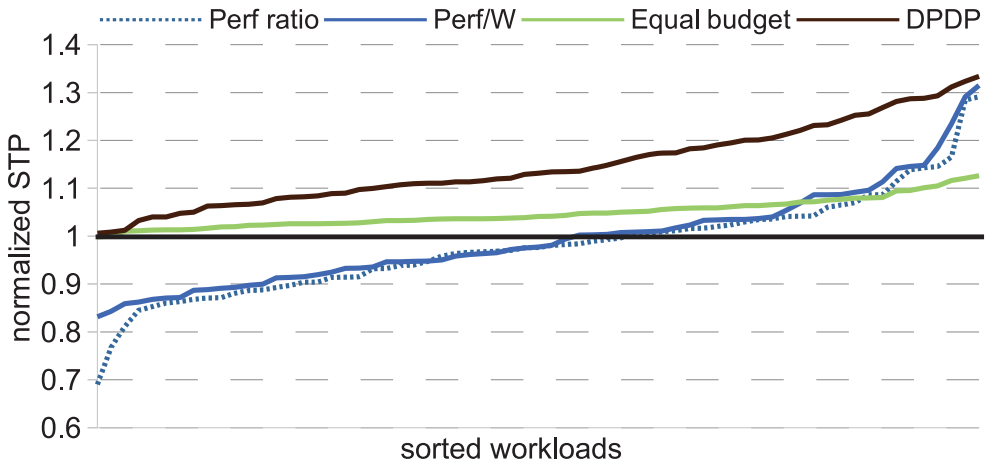


Fig. 10. Normalized STP for the various partitioning policies assuming a CMP configuration of 2 big and 4 little cores.

partitioning algorithm (Algorithm 1) also halts if all big cores are used before the budget runs out. Figure 10 shows the results for a CMP configuration consisting of four little cores and only two big cores. The general performance improvement over all other techniques is again clear. DPDP still shows a significant 14% improvement on average over global sprint-and-walk that reaches up to 33%. However, these gains are lower than our baseline results. Clearly, as the number of available big cores decreases, more applications are forced to stay on the little cores even when sufficient power budget is available.

6.4. Exploiting Application Phase Behavior

In DPDP, there is one application that runs on the big core for a fraction of the power averaging period and on the little core for the remaining fraction. This is done by first running on the big core, after which we switch to the little core. Further performance improvements could possibly be achieved by exploiting the phase behavior within an application. Some phases within the power-averaging period could benefit more from running on the big core than others. Therefore, instead of blindly running a fraction on the big core, we could carefully select the phases to run on the big core.

To assess the impact of selecting proper phases to run on the big core, we perform the following experiment. We simulate the dynamic sprint-and-walk scheme for each application on a single big–little pair (assuming a 1W per second budget). Furthermore, we simulate each application once on the big core and once on the little core, while collecting performance and power numbers for each 1M instruction interval. We then perform an exhaustive brute-force search to find the optimal schedule that optimizes performance within the budget by scheduling each 1M instruction interval either on the big or the little core. This optimal schedule assumes oracle knowledge of the phase behavior and does not incorporate migration overheads. We find that such an ideal scheduler improves performance by just 2% on average compared to dynamic sprint-and-walk, with only one application having more than 5% benefit.

This result is caused by multiple effects. First, the phase behavior within an application is less distinctive than the difference in behavior among applications. As such, optimizing the power budget with respect to phase behavior leads to a less significant improvement compared to optimizing the power budget across applications. Second, phases within an application that benefit from executing on the big core usually do not

match with the fraction of time in which the application should be executed on the big core as dictated by the power budget, giving less flexibility to the scheduler to fully exploit this phase behavior.

A realistic phase-aware scheduler would require an imperfect phase predictor and will suffer from migration overheads, which further reduces the potential benefits. Therefore, we decided not to implement a phase-aware scheduler to further improve the performance of the single application that migrates between the big and the little core.

6.5. Per-Application Performance Considerations

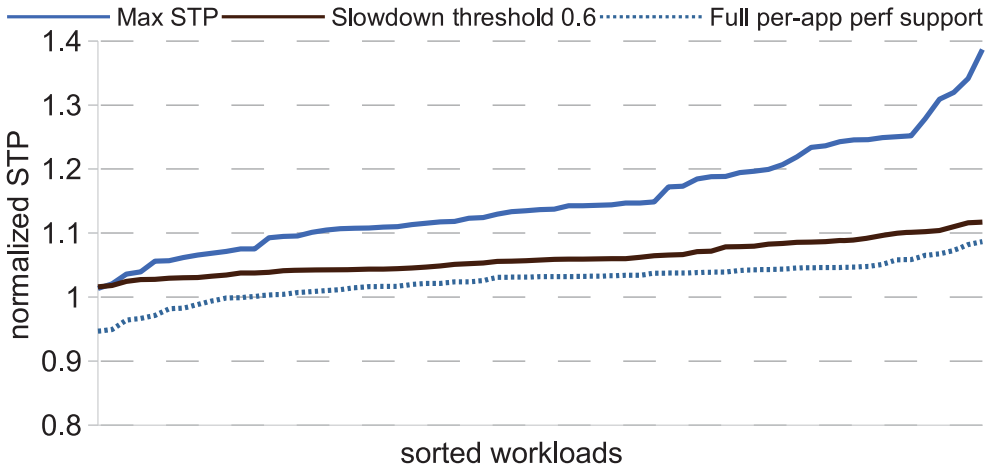
DPDP is designed to guarantee maximum STP under a power limit. To maximize throughput, DPDP favors applications with higher DP/DP values, giving them higher power budgets to run more on the big core. It is expected that applications with low DP/DP could suffer a slowdown compared to techniques that distribute power equally or that even greedily optimize the global power budget (e.g., global sprint-and-walk). In this section, we show how to extend DPDP with the capability to balance between the maximum throughput requirement and the maximum performance degradation that any single application suffers.

To avoid slowing down low-ranked applications too much, we allocate more power to those applications once a significant per-application performance degradation is detected. Our approach tries to control the degree of similarity by which applications progress in their execution (i.e., equal-progress fairness) [Van Craeynest et al. 2013]. To assess the progress of each application, we calculate the slowdown of each application using DPDP compared to always running on the big core. The similarity of slowdowns among applications indicates the fairness of the distribution, and reveals whether one application is suffering a relatively significant slowdown.

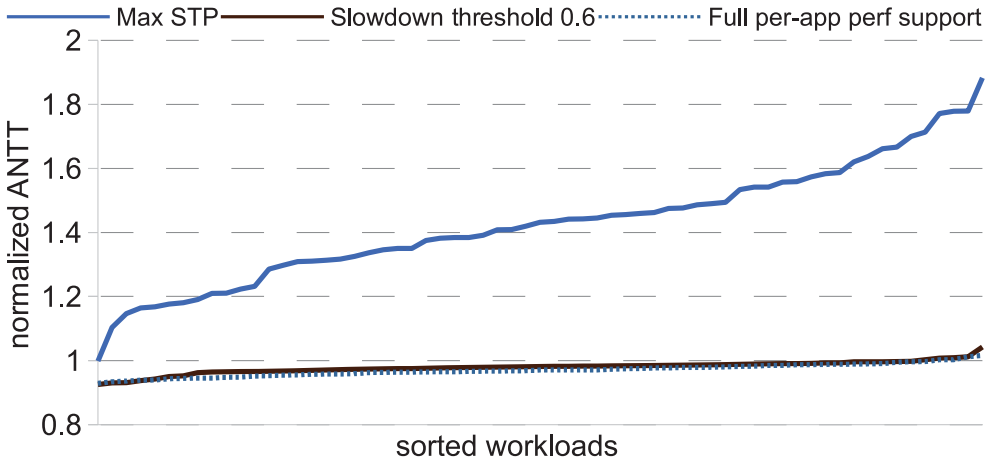
We use the ratio of the smallest slowdown to the highest slowdown among all applications to represent the equality of progress. We call this ratio the *progress index*. The closer this ratio to one, the fairer the power distribution and the progress of applications. The closer it is to zero, the higher the focus is on STP (i.e., lower regard to per-application slowdown). We provide the user with a knob to specify a slowdown threshold. At runtime, if the progress index drops below the slowdown threshold, DPDP focuses on improving per-application performance. Otherwise, it continues to maximize STP.

DPDP is a flexible power manager; thus, integrating this knob is straightforward. Every 100ms, DPDP reranks the applications based on their updated performance and power consumption. To provide per-application performance support, we calculate the progress index, in addition to the DP/DP metric, every 100ms. If the index drops below the specified threshold, DPDP reranks the applications based on their respective slowdowns. This ensures that applications with slower progress get more time on the big core over the next 100ms. When the progress index exceeds the slowdown threshold, DPDP resumes operating for maximum STP using the DP/DP metric.

Figure 11 shows the potential for adjusting the slowdown threshold to strike a sweet spot between STP and per-application performance. We use the ANTT to assess per-application performance. ANTT incorporates equal progress by heavily penalizing slowly running applications [Eyerman and Eeckhout 2008]. The higher ANTT normalized to global sprint-and-walk, the worse the impact of DPDP on the perceived per-application performance. When operating DPDP in full support of per-application performance (i.e., setting the slowdown threshold to 1), STP sees either insignificant gain or even a slight degradation for several mixes, as the dotted line in Figure 11(a) shows. On the other hand, when operating in maximum STP mode (i.e., original DPDP in which applications are always ranked using DP/DP), ANTT tends to increase by an



(a) System throughput for different per-application performance support thresholds



(b) ANTT for different per-application performance support thresholds

Fig. 11. STP and ANTT for different per-application performance support thresholds. A min-to-max slowdown point of 0.6 improves both STP (6%) and ANTT (3%).

average of 40%, as Figure 11(b) shows. By properly adjusting the slowdown threshold, DPDP is able to achieve similar ANTT to the mode of full per-application performance support (Figure 11(b)). ANTT gets slightly reduced compared to global sprint-and-walk, as can be seen for several mixes in the figure (mixes below 1). Figure 11(b) also shows that proper threshold tuning improves STP by 6% on average and up to 12%. More important, the results show the flexibility of DPDP to adapt to various system requirements.

7. RELATED WORK

We now discuss related work in power and thermal management, as well as recent work in scheduling for HCMPs.

7.1. Power and Thermal Management

Brooks and Martonosi [2001] discuss thermal constraints in microprocessors. They propose dynamic thermal management schemes for single-core processors using DVFS

and fetch throttling. Donald and Martonosi [2006] study dynamic thermal management for homogeneous multicores, and several papers [Cochran et al. 2011; Isci et al. 2006; Ma et al. 2011; Wang et al. 2009; Winter et al. 2010] propose schemes for maximizing the performance of homogeneous multicore processors under strict power limits using per-core DVFS. None of these DVFS works are directly applicable to HCMPs and neither do they consider the potential gains offered by temporarily exceeding the power cap.

Intel's Turboboost 1.0 [Gunther et al. 2010] increases the frequency when few cores are active, whereas Turboboost 2.0 [Rotem et al. 2012] allows for increasing the frequency beyond the TDP for short periods of time to improve responsiveness. Computational sprinting [Raghavan et al. 2012, 2013b] is a technique by which the responsiveness of interactive applications is improved by temporarily using more cores than the TDP allows, followed by an idle cool-down period. Our technique targets improving sustained chip throughput, rather than improving interactive responsiveness. Raghavan et al. [2013a] show that computational sprinting can also be beneficial for sustained performance if enabling more cores leads to a better energy efficiency. In a heterogeneous multicore setup, we find that a sprint-and-rest scheme (run on the big core, and then idle; the second technique in Figure 2) never outperforms a sprint-and-walk scheme (run on the big core, followed by running on the little core), because running on the little core is always more energy-efficient than running on the big core. Fan et al. [2016] describe an architecture to sprint data analytics applications at a rack level. They uses game theory to optimize STP of the whole rack given individual chip thermal limits and the rack-level power limit. An agent can sprint a chip by activating additional cores and raising their frequency. Their technique is not intended to partition the budget among multiple applications sharing the same chip. Our approach, on the other hand, takes a single chip running multiple applications concurrently. We improve STP by correctly selecting which applications to sprint on the big cores given a specific power budget.

Muthukaruppan et al. [2014] and Zhu et al. [2015] propose power and thermal management on HCMPs to improve energy efficiency while meeting QoS requirements. Here, we focus on optimal power management with maximizing total STP as a main objective. Paul et al. [2013] propose a technique to coordinate power and thermal management to improve performance and energy efficiency in systems consisting of both CPUs and GPUs.

7.2. Scheduling for Heterogeneous Multicores

Kumar et al. [2003] advocate single-ISA heterogeneous multicores to reduce power consumption. They show that a heterogeneous multicore is superior to DVFS in terms of energy efficiency. A recent study by Lukefahr et al. [2014] confirms that heterogeneity outperforms DVFS for low-power systems.

Many proposals advocate scheduling compute-intensive applications on big cores, because they show the highest performance improvement [Becchi and Crowley 2006; Chen and John 2009; Ghiasi et al. 2005; Koufaty et al. 2010; Shelepov et al. 2009]. Van Craeynest et al. [2012] show that memory-intensive applications can also show important performance gains on big cores if they are able to exploit more memory-level parallelism. All of these proposals optimize for performance or energy efficiency without considering power constraints. Our analysis shows that under power constraints, memory-intensive applications have another benefit: due to their lower power consumption, they can execute longer on big cores, which increases their overall performance, despite of their lower performance improvement on big cores.

8. CONCLUSIONS

Power and thermal constraints are becoming the main limiting factor in extracting high performance in modern processors. HCMPs provide flexibility to improve

performance under power limits: if power headroom is available, applications can execute on big, powerful cores, while executing on little, energy-efficient cores cools down the chip and builds up new headroom. This article explores mechanisms to maximize the performance of an HCMP under power limits.

We show that global greedy scheduling or equal budget partitioning schemes do not lead to optimal performance, because some applications can use the budget more efficiently than others. Previously proposed scheduling schemes for performance and energy efficiency also do not reach optimal performance because they ignore the fraction of time that applications can make use of the big core as implied by the power budget. Using linear programming, we deduce that ranking applications by their DP/DP ratio leads to the theoretically optimal schedule.

We propose and evaluate a scheduler that uses the DP/DP metric, and show that it outperforms the other schedulers by a significant margin. Our experimental results with 4 big.LITTLE pairs demonstrate that DPDP outperforms global greedy scheduling, a natural translation of Intel's TurboBoost to HCMPs, by 16% on average and up to 40%. An interesting observation is that, under power constraints, it is beneficial to favor memory-intensive applications to run on the big core, whereas prior work advocates scheduling compute-intensive applications on the big core (in the absence of power constraints). The reason for this counterintuitive result is that memory-intensive applications usually consume less power on the big core, allowing them to run on the big core for a longer period of time, thereby improving overall performance within the power budget.

REFERENCES

Michela Becchi and Patrick Crowley. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*. 29–40.

David Brooks and Margaret Martonosi. 2001. Dynamic thermal management for high-performance microprocessors. In *7th International Symposium on High-Performance Computer Architecture (HPCA'01)*. 171–182.

Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization* 11, 3, 28.

Jian Chen and Lizy K. John. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. 927–930.

N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, Li Zhao, N. Ijih, S. Subhaschandra, S. Grover, Xiaowei Jiang, and R. Iyer. 2012. QuickIA: Exploring heterogeneous architectures on real prototypes. In *18th International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–8.

Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & cap: Adaptive DVFS and thread packing under power caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. 175–185.

James Donald and Margaret Martonosi. 2006. Techniques for multicore thermal management: Classification and new exploration. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 78–88.

H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *38th Annual International Symposium on Computer Architecture (ISCA'11)*. 365–376.

Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3, 42–53.

Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. 2016. The computational sprinting game. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 561–575.

Soraya Ghiasi, Tom Keller, and Freeman Rawson. 2005. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*. 199–210.

Peter Greenhalgh. 2011. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. ARM White paper.

Steve Gunther, Anant Deval, Ted Burton, and Rajesh Kumar. 2010. Energy-efficient computing: Power management system on the Nehalem family of processors. *Intel Technology Journal* 14, 3.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward dark silicon in servers. *IEEE Micro* 31, 6–15.

Scott Huck. 2011. Measuring processor power. Intel white paper.

Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. 2006. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO'06)*. 347–358.

Brian Jeff. 2013. big.LITTLE Technology moves towards fully heterogeneous global task scheduling. ARM White paper.

David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*. 125–138.

Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *36th International Symposium on Microarchitecture (MICRO'03)*. 81–92.

Belli Kuttana. 2013. Technology Insight: Intel Silvermont Microarchitecture. Intel Developer Forum.

Nagesh B. Lakshminarayana, Jaekyu Lee, and Hyesoon Kim. 2009. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 25.

Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2008. Power capping: A prelude to power shifting. *Cluster Computing* 11, 2, 183–195.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd International Symposium on Microarchitecture (MICRO'09)*. 469–480.

Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski Jr, Thomas F. Wenisch, and Scott Mahlke. 2014. Heterogeneous microarchitectures trump voltage scaling for low-power cores. In *23rd International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 237–250.

Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. 2012. Composite cores: Pushing heterogeneity into a core. In *45th International Symposium on Microarchitecture (MICRO'12)*. 317–328.

Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *38th Annual International Symposium on Computer Architecture (ISCA'11)*. 449–460.

Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price theory based power management for heterogeneous multi-cores. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 161–176.

NVIDIA. 2011. Variable SMP – A multi-core CPU architecture for low power and high performance. White paper.

Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'04)*. 81–92.

Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. 2013. Cooperative boosting: Needy versus greedy power management. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. 285–296.

Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2013a. Utilizing dark silicon to save energy with computational sprinting. *IEEE Micro* 33, 5, 20–28.

Arun Raghavan, Laurel Emurian, Lei Shao, Marios C. Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2013b. Computational sprinting on a hardware/software testbed. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 155–166.

Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios C. Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2012. Computational sprinting. In *18th International Symposium on High Performance Computer Architecture (HPCA'12)*. 249–260.

Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. 2012. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro* 2, 20–27.

Samsung Electronics. 2013. Samsung Primes Exynos 5 Octa for ARM big.LITTLE Technology with Heterogeneous Multi-Processing Capability. Press release.

Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. HASS: A scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review* 43, 2, 66–75.

Michael B. Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. 1131–1136.

Michael B. Taylor. 2013. A landscape of the new dark silicon design regime. *IEEE Micro* 33, 5, 8–19.

Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. 177–187.

Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *International Symposium on Computer Architecture (ISCA'12)*. 213–224.

Yefu Wang, Kai Ma, and Xiaorui Wang. 2009. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 314–324.

Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. 2010. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 29–40.

Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. 2015. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *21st International Symposium on High Performance Computer Architecture (HPCA)*. 137–149.

Received May 2016; revised July 2016; accepted July 2016