2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Asymmetry-Aware Work-Stealing Runtimes

Christopher Torng, Moyang Wang, and Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{clt67,mw828,cbatten}@cornell.edu

***Abstract*—Amdahl's law provides architects a compelling reason to introduce system asymmetry to optimize for both serial and parallel regions of execution. Asymmetry in a multicore processor can arise statically (e.g., from core microarchitecture) or dynamically (e.g., applying dynamic voltage/frequency scaling). Work stealing is an increasingly popular approach to task distribution that elegantly balances task-based parallelism across multiple worker threads. In this paper, we propose asymmetry-aware work-stealing (AAWS) runtimes, which are carefully designed to exploit both the static and dynamic asymmetry in modern systems. AAWS runtimes use three key hardware/software techniques: work-pacing, work-sprinting, and work-mugging. Work-pacing and work-sprinting are novel techniques that combine a marginal-utility-based approach with integrated voltage regulators to improve performance and energy efficiency in high- and low-parallel regions. Work-mugging is a previously proposed technique that enables a waiting big core to preemptively migrate work from a busy little core. We propose a simple implementation of work-mugging based on lightweight user-level interrupts. We use a vertically integrated research methodology spanning software, architecture, and VLSI to make the case that holistically combining static asymmetry, dynamic asymmetry, and work-stealing runtimes can improve both performance and energy efficiency in future multicore systems.**

I. Introduction

Work stealing is a well-known approach to task distribution that elegantly balances task-based parallelism across multiple worker threads [10, 30]. In a work-stealing runtime, each worker thread enqueues and dequeues tasks onto the tail of its task queue. When a worker finds its queue empty, it attempts to steal a task from the head of another worker thread's task queue. Work stealing has been shown to have good performance, space requirements, and communication overhead in both theory [8] and practice [7,22]. Optimizing work-stealing runtimes remains a rich research area [4,12,13,15,18,47], and work stealing is a critical component in many popular concurrency platforms including Intel's Cilk++, Intel's C++ Threading Building Blocks (TBB), Microsoft's .NET Task Parallel Library, Java's Fork/Join Framework, X10, and OpenMP. Most of the past research and current implementations use *asymmetry-oblivious work-stealing* runtimes. In this work, we propose *asymmetry-aware work-stealing* (AAWS) runtimes, which exploit both *static asymmetry* (e.g., different core microarchitectures) and *dynamic asymmetry* (e.g., per-core dynamic voltage/frequency scaling) to improve the performance and energy efficiency of multicore processors.

Single-ISA heterogeneous processors integrate multiple cores with different microarchitectures onto a single die in order to provide distinct energy-performance operating points [42, 43]. These processors exhibit *static asymmetry* that is fixed at design time. Systems based on ARM's big.LITTLE architecture, which composes "big" ARM Cortex-A15/A57 out-of-order cores with "little" Cortex-A7/A53 in-order cores [25, 41], are commercially available from Samsung [29], Qualcomm [28], Mediatek [17], and Renesas [27]. There has been significant interest in new techniques for effectively scheduling software across these big and little cores, although most of this prior work has focused on either multiprogrammed workloads [1, 42, 43, 60] or on applications that use thread-based parallel programming constructs [35, 36, 44, 59]. However, there has been less research exploring the interaction between state-of-the-art work-stealing runtimes and static asymmetry. A notable exception is Bender et al.'s theoretical work [3] and abstract discrete-event modeling [37] on an enhanced Cilk scheduler for heterogeneous systems. Others have also explored combining work-stealing with work-sharing, critical-path scheduling, and/or core affinity to more effectively schedule tasks across statically asymmetric systems [13, 14, 16, 52].

Dynamic voltage/frequency scaling (DVFS) is an example of *dynamic asymmetry* that is adjustable at runtime. Much of the prior work on DVFS has assumed off-chip voltage regulation best used for coarse-grain voltage scaling [9, 32, 33]. Recent architecture/circuit co-design of fine-grain DVFS (either through multi-rail voltage supplies [19, 51] or fully integrated voltage regulators [21, 23, 24, 34, 40, 58]) suggests that sub-microsecond voltage transition times may be feasible in the near future. There has been significant interest in new techniques for exploiting fine-grain DVFS to improve the performance and/or energy efficiency of multiprogrammed workloads [20, 40] or on applications that use thread-based parallel programming constructs [4, 11, 24, 48, 51]. Again, there has been relatively little research exploring the interaction between work-stealing runtimes and dynamic asymmetry. A notable exception is recent work by Ribic et al. that proposes reducing the voltage/frequency of thieves and increasing the voltage/frequency of workers with deep task queues [55].

Recent studies have demonstrated the potential benefit of combining static and dynamic asymmetry [2, 26, 50]. A key observation is that static asymmetry through heterogeneous core types offers larger marginal utility but must be fixed at design time, while dynamic asymmetry in the form of DVFS offers smaller marginal utility but can be varied at run time [2]. These past works have focused on coarse-grain multiprogrammed workloads. To our knowledge, this is the first work to explore the interaction between static asymmetry, dynamic asymmetry, and work-stealing runtimes. We argue that work-stealing runtimes are a natural fit for managing asymmetry. Assuming fully strict programs with high-parallel slack [22], a work-stealing runtime will naturally exploit asymmetry without modification; faster cores will execute tasks more quickly and then simply steal work as necessary from slower cores. However, our work shows there are

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.14

still important opportunities for AAWS runtimes to improve performance and energy efficiency.

In Section II, we develop a simple first-order model to provide insight into optimizing the aggregate throughput and energy efficiency of an AAWS runtime. Our model predicts that the maximum throughput will occur when the marginal utility (i.e., performance) vs. marginal cost (i.e., power) of each core is equal. This is an intuitive application of a fundamental principle in economics known as the *Law of Equi-Marginal Utility*. Others have also observed this important design guideline in the context of design-space exploration of processor microarchitecture [2] and market-based multi-resource allocation in multicore processors [62]. We use numerical analysis to study the potential benefit of a marginal-utility-based approach in AAWS runtimes for both high-parallel (HP) and low-parallel (LP) regions.

In Section III, we propose three new hardware/software mechanisms to enable AAWS runtimes: *work-pacing*, *work-sprinting*, and *work-mugging*. Work-pacing is a novel technique that directly applies a marginal-utility-based approach in the HP region by increasing the voltages of little cores and decreasing the voltages of big cores to improve both performance and energy efficiency. Work-sprinting is a novel technique similar to work-pacing that applies a marginal-utility-based approach in the LP region, while also exploiting additional power slack generated from resting waiting cores. Work-pacing and work-sprinting require a fine-grain DVFS controller specifically designed for an AAWS runtime as well as lightweight hint instructions that allow the AAWS runtime to inform the hardware when cores are active or waiting. While work-sprinting can provide some benefit in LP regions, it is also fundamentally limited by the work-stealing scheduler's task distribution algorithm. Our first-order modeling suggests that when distributing power among busy cores in the LP region, sprinting a little core while resting a big core is usually suboptimal compared to resting a little core and sprinting a big core. To address this issue, we revisit previous theoretical work on work-mugging [3, 37]. Work-mugging allows a waiting big core to "mug" an active little core by preemptively migrating a task from the little core to the big core. We describe a practical implementation of work-mugging in multicore systems that relies on user-level inter-core interrupts.

In Section IV, we outline our vertically integrated research methodology. We have developed our own C++ work-stealing runtime inspired by Intel TBB which uses Chase-Lev task queues [12] and occupancy-based victim selection [15]. We have ported 20 application kernels to our work-stealing runtime. These kernels are selected from the PBBS [57], PARSEC [5], Cilk [22], and UTS [53] benchmark suites and represent diverse application-level characteristics. We evaluate the performance of two eight-core systems (4 big & 4 little; 1 big & 7 little) using the cycle-level gem5 simulator [6], and we evaluate power and energy efficiency using a detailed power/energy model that leverages component models within McPAT [49] as well as energy estimates from our own VLSI implementation of a single-issue in-order RISC processor.

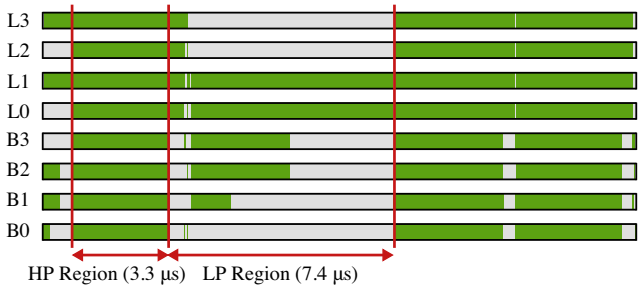


Figure 1. Activity Profile for Convex Hull Application on Statically Asymmetric System – Only a subset of the entire activity profile is shown. Cores L0–3 are "little" in-order cores; cores B0–3 are "big" out-of-order cores. Green = executing task; light-gray = waiting in work-stealing loop; HP = high-parallel; LP = low-parallel. See Section IV for simulation methodology.

In Section V, we use this methodology to explore the potential performance and energy efficiency benefits of AAWS runtimes. On a system with four big and four little cores, an AAWS runtime achieves speedups from 1.02–1.32× (median: 1.10×). At the same time, all but one kernel achieves improved energy efficiency with a maximum improvement of 1.53× (median: 1.11×).

The key contributions of this work are: (1) we develop a marginal-utility-based approach to both quantify the potential benefit of AAWS runtimes and motivate specific hardware/software techniques; (2) we propose new work-pacing and work-sprinting techniques that directly apply a marginal-utility-based approach within AAWS runtimes; (3) we provide a practical implementation of the previously proposed work-mugging technique suitable for use in AAWS runtimes; and (4) we use a vertically integrated research methodology that spans software, architecture, and VLSI to make the case that holistically combining static asymmetry, dynamic asymmetry, and work-stealing runtimes can improve both performance and energy efficiency in future multicore systems.

II. A Marginal-Utility-Based Approach

Figure 1 shows an activity profile for an example application running on a system with four big cores and four little cores and an asymmetry-oblivious work-stealing runtime. Notice that the application includes a mix of both high-parallel (HP) and low-parallel (LP) regions. During HP regions, the work-stealing runtime is able to adapt to the static asymmetry by distributing more tasks to the bigger cores resulting in good throughput and a relatively balanced profile. During LP regions, some cores are active, while other cores are waiting in the work-stealing loop. Notice that there is usually a small LP region near the end of an HP region, since the work-stealing runtime is unable to redistribute work at infinitely small granularities.

AAWS runtimes attempt to improve performance and energy efficiency in both the HP and LP regions by making the work-stealing runtime aware of the underlying static and dynamic asymmetry. In this section, we use first-order modeling and numerical analysis (similar to [31,63]) to motivate the three techniques we will explore in this paper: *work-pacing*,

which targets the HP region; *work-sprinting*, which focuses on the LP region; and *work-mugging*, which also focuses on the LP region.

A. First-Order Model

Consider a multicore system comprised of N_B big cores and N_L little cores, with N_{BA}, N_{BW}, N_{LA}, N_{LW} big/little active/waiting cores. Assume both big and little cores have the same nominal voltage (V_N) and nominal frequency (f_N), and that the system can individually scale the voltage of each core from V_{min} to V_{max}. Waiting cores can execute the work-stealing loop while "resting" at V_{min} to save power while still enabling correct execution.

We assume that frequency is a linear function of voltage (validated using circuit-level simulation, see Section IV). The frequency of each active core is thus:

$$\begin{aligned} f_{Bi} &= k_1 V_{Bi} + k_2 \qquad (i = 1,2,...,N_{BA}) \\ f_{Lj} &= k_1 V_{Lj} + k_2 \qquad (j = 1,2,...,N_{LA}) \end{aligned} \tag{1}$$

where k_1, k_2 are fitted parameters, f_{Bi} is the frequency of big core i, V_{Bi} is the voltage of big core i, and so on.

The throughput of an active core is measured in instructions per second (*IPS*) and is a function of the average instructions per cycle (*IPC*) of a given core type:

$$\begin{aligned} IPS_{BAi} &= IPC_B\ f_{Bi} \qquad (i = 1,2,...,N_{BA}) \\ IPS_{LAj} &= IPC_L\ f_{Lj} \qquad (j = 1,2,...,N_{LA}) \end{aligned} \tag{2}$$

To simplify our discussion we define $\beta = IPC_B/IPC_L$.

We use the aggregate throughput of all active cores as an approximation for the performance of the overall application. If we assume compute-bound tasks and perfect task-balancing through work-stealing in the HP region, then increasing the total throughput will indeed reduce the overall execution time of the HP region. The performance of the LP region is more subtle, since by definition the LP region cannot take advantage of work-stealing until more work is generated. Increasing the throughput of one core at the expense of another core may or may not improve execution time depending on when cores reach the next synchronization point. Fortunately, resting waiting cores in the LP region can generate power slack that can be reallocated to the active cores. This means in practice, we are usually increasing the performance of all active cores in the LP region, and thus using the aggregate throughput can still provide useful insight into how to scale the relative voltages of each core. Given these caveats, we model the total performance of the multicore system as:

$$IPS_{tot} = \sum_{i=1}^{N_{BA}} IPS_{BAi} + \sum_{j=1}^{N_{LA}} IPS_{LAj} \tag{3}$$

The core power includes both dynamic and static power and is modeled as:

$$\begin{aligned} P_{BAi} &= \alpha_B\ IPC_B\ f_{Bi}\ V_{Bi}^2 + V_{Bi}\ I_{B,leak} \qquad (i = 1,2,...,N_{BA}) \\ P_{LAj} &= \alpha_L\ IPC_L\ f_{Lj}\ V_{Lj}^2 + V_{Lj}\ I_{L,leak} \qquad (j = 1,2,...,N_{LA}) \end{aligned} \tag{4}$$

The factors α_B and α_L capture the relative energy overhead of a big core compared to a little core. To simplify our discussion, we define $\alpha = \alpha_B/\alpha_L$ as the energy ratio of a big core to a little core at nominal voltage and frequency. We calculate the leakage current by assuming an architect targets leakage power to consume a certain percentage (denoted as λ) of the total power of a big core at nominal voltage. We assume a little core's leakage current is a fraction (denoted by γ) of the big core's leakage current. We use P_{BN}, P_{LN}, P_{BW}, and P_{LW} to refer to the power consumed by big and little cores running at nominal V_N or waiting at V_{min}.

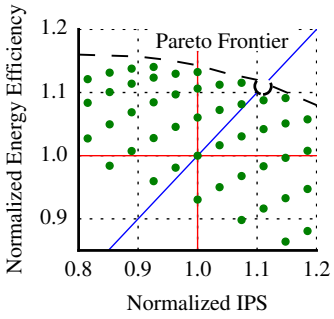


Figure 2. Pareto-Optimal Frontier for 4B4L System – Projected energy efficiency vs. performance of a busy 4B4L system across different (V_{Bi}, V_{Lj}) pairs. Points normalized to (1.0 V, 1.0 V) system. Diagonal line is isopower. Open circle = pareto-optimal isopower system. $\alpha = 3$, $\beta = 2$.

The total power is the aggregate power across all cores:

$$P_{total} = \sum_{i=1}^{N_{BA}} P_{BAi} + \sum_{j=1}^{N_{LA}} P_{LAj} + N_{BW}\ (P_{BW}) + N_{LW}\ (P_{LW}) \tag{5}$$

B. Marginal Utility Optimization Problem

In Figure 2, we use our first-order model to generate energy vs. performance estimates for a system with four big and four little cores (denoted as 4B4L). The array of plotted points represent selections of different (V_{Bi}, V_{Lj}) pairs, with all estimates normalized to the nominal system (1.0 V, 1.0 V). We assume that all cores are busy with useful work. Points in the lower-right quadrant generally have higher voltage and frequency, suggesting that performance can easily be improved at the expense of energy efficiency and at higher power. Points in the upper-left quadrant generally have lower voltage and frequency, suggesting that energy efficiency can easily be improved at the expense of performance with lower power. However, points in the upper-right quadrant suggest that careful tuning of the voltages and frequencies of the big and little cores can potentially improve performance *and* energy efficiency at the same time.

Our goal is to create an optimization problem to find the pareto-optimal points in the upper-right quadrant. As a basic heuristic, we use average system power at nominal voltage to constrain our optimization problem, effectively targeting a pareto-optimal system that draws similar power compared to the nominal system (denoted by the open circle). The target system power is therefore the aggregate power of all cores running at nominal voltage and frequency:

$$P_{target} = N_B\ (P_{BN}) + N_L\ (P_{LN}) \tag{6}$$

More specifically, our optimization problem searches for the optimal voltages for active big (V_{Bi}) and little (V_{Li}) cores such that the total throughput (IPS_{tot}) is maximized while maintaining the power target (P_{target}). We use the method of Lagrange multipliers to solve this optimization problem,

and we rewrite the final result in terms of the marginal performance vs. marginal power as follows:

$$\frac{\partial P_{BAi}}{\partial IPS_{BAi}} = \frac{\partial P_{LAj}}{\partial IPS_{LAj}} (i = 1,2,...,N_{BA}; j = 1,2,...,N_{LA}) \quad (7)$$

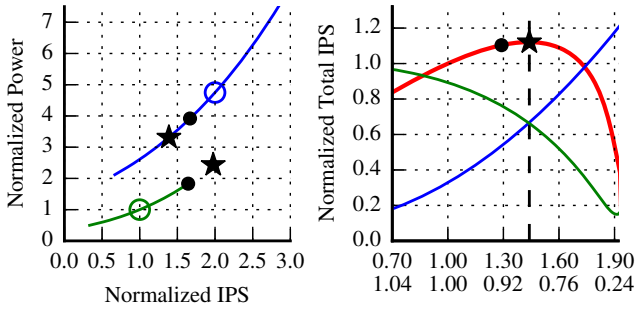
This is an intuitive application of a fundamental principle in economics known as the *Law of Equi-Marginal Utility*. At the optimum operating point the marginal utility (i.e., performance) vs. marginal cost (i.e., power) of each core must be equal. If this was not the case, then an arbitrage opportunity would exist: we could "sell" expensive performance to reclaim power on one core and "buy" more performance at a cheaper price (power) on another core. Others have also recognized that the Law of Equi-Marginal Utility provides an elegant digital design principle [2, 62], although here we are applying this principle in a slightly different context.

Unfortunately, a closed-form solution for the optimum V_{Bi} and V_{Lj} can be complex, so in the remainder of this section we use numerical analysis to explore using a marginal-utility-based approach in both the HP and LP regions. Unless otherwise noted we will assume the following parameters: k_1 = 7.38×10^8, k_2 = -4.05×10^8, V_N = 1 V, V_{min} = 0.7 V, V_{max} = 1.3 V, f_N = 333 MHz, λ = 0.1, γ = 0.25. These parameters are derived from VLSI modeling for the target voltage range and system described in Section IV.

C. Marginal Utility in the High-Parallel Region

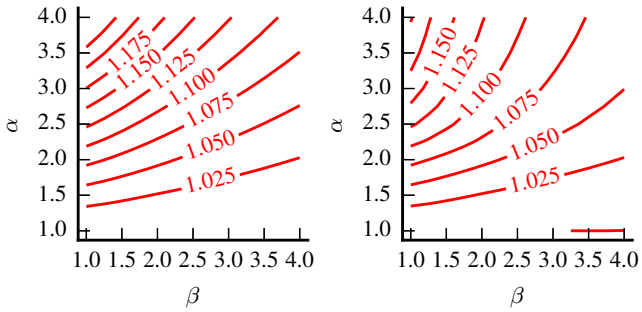
Figure 3 uses the first-order model developed in the previous subsections to plot the power and performance of a 4B4L system. This is a common configuration found in commercially available ARM big.LITTLE systems [28, 29]. We can immediately see the benefit of static asymmetry in Figure 3(a). The big core offers higher performance at higher power, while the little core offers lower performance at lower power. Figure 3(b) shows the marginal utility of the big core (blue curve) and little core (green curve) as well as IPS_{tot}. As expected, IPS_{tot} is maximized when the marginal utilities are equal. The optimal operating point is V_{Bi} = 0.86 V and V_{Li} = 1.44 V with a theoretical speedup of 1.12× over running all cores at V_N. Since 1.44 V > V_{max}, the best feasible operating point is V_{Bi} = 0.93 V and V_{Li} = V_{max} with a theoretical speedup of 1.10×. Figure 4 shows how the optimal and feasible speedup varies as a function of α and β. A marginal-utility-based approach is most effective when the big core has moderate performance benefit for large energy overhead (i.e., $\alpha/\beta > 1.0$), which matches the conventional wisdom concerning big vs. little cores. This wisdom is supported by data collected by ARM during pre-silicon design-space exploration of Cortex-A7 and Cortex-A15 cores [25] as well as by our own results (see Section V).

This analysis suggests a marginal-utility-based approach can offer respectable speedups in the HP region, and thus motivates our interest in our new work-pacing technique. It is important to note, that a marginal-utility-based approach requires holistically considering static asymmetry, dynamic asymmetry, *and* a work-stealing runtime. With a thread-based parallel programming framework instead of work-



(a) Power vs. IPS (b) Total IPS vs. V_{Bi} and V_{Lj}

Figure 3. 4B4L System w/ All Cores Active – (a) Power vs. performance curves across the DVFS operating points for each core type, green = little, blue = big, circle = nominal; (b) blue = $\partial P_{BAi}/\partial IPS_{BAi}$ (axis not shown), green = $\partial P_{LAj}/\partial IPS_{LAj}$ (axis not shown), red = IPS_{tot} (axis on left) assuming V_{Lj} and V_{Bi} shown on x-axis (V_{Lj} on top, V_{Bi} on bottom) with constant P_{target}. (a–b) star = optimal operating point, dot = feasible operating point, α = 3, β = 2.



(a) Optimum Speedup (b) Feasible Speedup

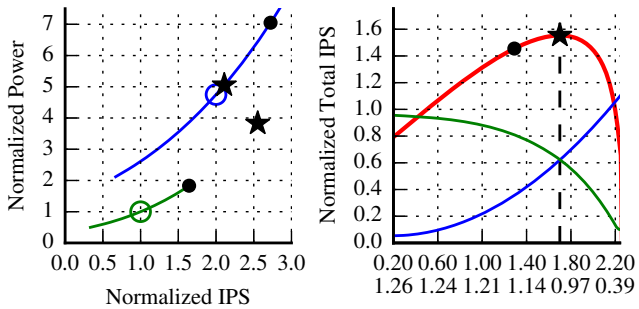
Figure 4. Theoretical Speedup for 4B4L System vs. α and β – (a) optimum speedup ignoring V_{min} and V_{max}; (b) = feasible speedup within V_{min} and V_{max}. Speedups relative to all cores running at V_N.

stealing, slowing down the big core would likely create a significantly lagging thread hurting the overall execution time. Without static asymmetry, the Law of Equi-Marginal Utility tells us that the optimal approach is to simply run all homogeneous cores at V_N during the HP region. Without dynamic asymmetry, there is no opportunity to adaptively "trade" performance vs. power and thus no way to balance the marginal utilities in the HP region.

D. Marginal Utility in the Low-Parallel Region

Figure 5 plots the power and performance of a 4B4L system in the LP region with two active big cores and two active little cores. We can rest the waiting cores, generating power slack that can then be reallocated to the active cores. The resulting optimal operating point is V_{Bi} = 1.02 V and V_{Li} = 1.70 V with a theoretical speedup of 1.55× over running all cores at V_N. Obviously running the little core at 1.70 V is not feasible, so the best feasible operating point is V_{Bi} = 1.16 V and V_{Li} = V_{max} with a theoretical speedup of 1.45×.

Note that we can potentially further improve performance by moving tasks from little to big cores. As shown in Figure 5(a) the little core often reaches V_{max} before it can com-



(a) Power vs. IPS (b) Total IPS vs. V_{Bi} and V_{Lj}

Figure 5. 4B4L System w/ 2B2L Active – Assume we rest inactive cores at V_{min}. See Figure 3 for legend. $\alpha = 3$, $\beta = 2$.

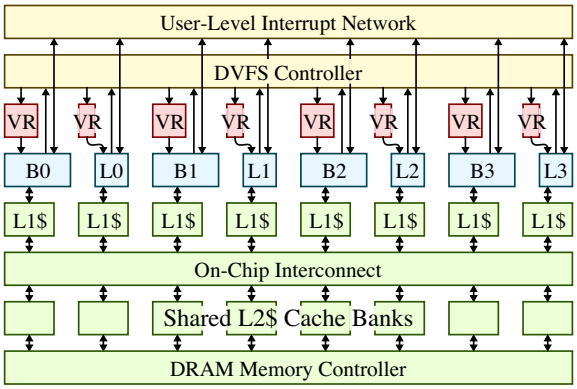


Figure 6. 4B4L System w/ Hardware Support for AAWS Runtimes – Work-pacing and work-sprinting require fully integrated voltage regulators for fine-grain DVFS (i.e., per-core, sub-microsecond scaling to potentially many voltage levels) as well as a customized DVFS controller. Work-mugging requires user-level interrupts to enable rapid communication between cores. B0–B3 = big cores, L0–L3 = little cores, VR = fully integrated voltage regulator.

pletely exploit the power slack generated from resting cores. Moving tasks to a big core increases peak performance and thus can help accelerate LP regions. As an example, assume there is a single remaining task in a 4B4L system and we must decide whether to execute this task on a little or big core. Our first-order model predicts that the optimum operating point when using a little core is $V_L = 2.59\,\text{V}$, but the feasible operating point is V_{max} with a theoretical speedup of 1.6× over running this task on the little core at V_N. If we instead move this final task to a big core, the optimum operating point is $V_B = 1.51\,\text{V}$, and the feasible operating point is again V_{max} with a theoretical speedup of 3.3× over running this task on the little core at V_N. Moving work from little to big cores in the LP range can significantly improve performance if we take into account the feasible voltage range.

This analysis suggests that a marginal-utility-based approach can be useful in the LP region, but our analysis also motivates our interest in a practical implementation of work-mugging. Work-mugging can preemptively move work from little to big cores, and thus helps keep big cores busy during the LP region. Again a holistic approach is required: using just dynamic or static asymmetry during the LP region is unlikely to fully exploit the generated power slack.

III. AAWS Runtimes

In this section, we describe how AAWS runtimes can use three new hardware/software techniques: *work-pacing*, *work-sprinting*, and *work-mugging*. We also describe two simpler techniques, *serial-sprinting* and *work-biasing*, which we include in our aggressive baseline runtime.

A. Work-Pacing and Work-Sprinting

Work-pacing uses a marginal-utility-based approach to maximize throughput in the HP region by increasing the voltage of little cores and decreasing the voltage of big cores. Work-sprinting combines the power slack generated from resting waiting cores in the LP region with a marginal-utility-based approach to again maximize throughput. An efficient implementation of work-pacing and work-sprinting requires lightweight changes to both the hardware and software.

Work-pacing and work-sprinting require the AAWS software runtime to inform the hardware of when threads are either actively executing tasks or waiting in the work-stealing loop. We propose instrumenting the work-stealing loop in the AAWS runtime with hint instructions, such that each hint instruction toggles an activity bit indicating the status of each core. This is similar to past work on lightweight DVFS controllers for applications that use thread-based parallel programming frameworks [24, 51]. When a worker thread enters the work-stealing loop, it will wait until its second steal attempt before using the hint instruction to toggle the activity bit. When combined with occupancy-based victim selection [15] as opposed to random victim selection [22], this avoids unnecessary activity bit transitions that could adversely impact the customized DVFS controller described later in this subsection. Although this elegant approach is reactive, it also avoids the need for prediction heuristics [4, 11].

As shown in Figure 1, HP and LP region timescales can be on the order of a few microseconds. Unfortunately, traditional off-chip switching regulators can have voltage scaling response times on the order of tens to hundreds of microseconds [9, 54]. Multi-rail voltage supplies offer very fast response times on the order of tens of nanoseconds [19,51], but they offer limited voltage levels making them a poor fit for the many voltage levels required by our marginal-utility-based approach. We propose leveraging recent circuit-level research on fully-integrated switching regulators [39, 45, 46]. Architects are increasingly making the case for integrated voltage regulation as a way to reduce system cost and enable fine-grain DVFS for per-core, sub-microsecond scaling to potentially many voltage levels [23, 24, 34, 40, 64]. Indeed, the Intel Haswell processor uses in-package inductors with on-chip regulators to provide fast-changing per-core supplies [38].

Figure 6 illustrates a 4B4L system that can potentially leverage an AAWS runtime. Each core has its own private L1 instruction and data cache along with a shared, banked L2 cache. The activity bits controlled by the hint instructions are read by a global DVFS controller, which then decides how to set the supply voltages of fully integrated per-core voltage

regulators. While distributed DVFS controllers are certainly possible [62], the added complexity is probably not warranted for smaller-scale systems. We propose using a simple lookup-table-based DVFS controller to map activity information into appropriate voltage levels. The marginal-utility-based analysis from the previous section along with estimated values for α and β can be used to create a lookup table that performs generally well across a wide range of application kernels. For a 4B4L system, there are five possible values for the number of active little (big) cores, including zero. The lookup table would therefore include 25 entries. More sophisticated adaptive algorithms that update the lookup tables based on performance and energy counters are possible and an interesting direction for future work.

B. Work-Mugging

The goal of work-mugging is to preemptively migrate work from little cores to big cores during the LP region. Our first-order modeling argues for using the big cores when possible during the LP region, since bigger cores have a higher feasible performance limit. Previous theoretical work [3] and abstract discrete-event modeling [37] have made the case for work-mugging, although these past works did not propose a concrete way to actually implement work-mugging. As with work-pacing and work-sprinting, an efficient implementation of work-mugging requires lightweight changes to both the hardware and software.

We propose using fast user-level interrupts to enable one core to redirect the control flow of another core. These user-level interrupts are essentially a simple implementation of previous ideas on asynchronous direct messages [56] and active messages [61]. A "mugger" can use a new `mug` instruction to mug a "muggee". The `mug` instruction takes two input values stored in registers. One indicates which core to mug and the other contains the address of the user-level interrupt handler. Other than the handler address, all data is communicated through shared memory. This form of user-level interrupts requires a simple, low-bandwidth inter-core network with approximately four-byte messages (see Figure 6).

We modify the AAWS runtime so that when a worker thread running on a big core enters the work-stealing loop, it will attempt to steal work twice before considering work-mugging. If at least one little core is active, then the big core selects a little core to mug using the `mug` instruction. Each core's runtime keeps information about that core's activity and task queue occupancy in shared memory, so that every core can easily determine which cores are active and which cores are waiting. The mugger and muggee store their user-level architectural state to shared memory, synchronize at a barrier, and then load the other thread's user-level architectural state. The overall effect is that a big core begins executing a task which was previously executing on a little core, while the little core enters the work-stealing loop. A subtle yet critical implementation detail involves ensuring that a big core always executes the sequential part of the program after the parallel region (i.e., logical thread 0). Otherwise, a little core can end up executing the serial region. We modify the AAWS runtime so at the end of a parallel region, logical thread 0 checks to see if it is running on a big core. If not, it simply uses a `mug` instruction to mug any big core.

While work-mugging can help preemptively move work from little to big cores, it can also cause additional L1 cache misses as the task's working set is gradually migrated. This overhead is somewhat mitigated by the good locality properties of a work-stealing runtime and by the fact that little cores do not mug work back from a big core during the LP region.

C. Serial-Sprinting and Work-Biasing

Instruction hints can also be used to inform the hardware of truly serial regions (as opposed to an LP region which just happens to have one task remaining). An obvious extension is to allow the DVFS controller to sprint the single active big core during these serial regions. Since this is a relatively straight-forward optimization, we include *serial-sprinting* in our baseline work-stealing runtime. Our application workloads have relatively short serial regions. So while serial-sprinting does not hurt performance, it also does not offer much benefit (approximately 1–2%).

A simple non-preemptive scheme we call *work-biasing* involves preventing little cores from stealing work unless all big cores are active. Work-biasing has the effect of "biasing" work towards the big cores, but the non-preemptive nature of work-biasing means there are few opportunities to have a meaningful impact on the work distribution. Indeed, the prior theoretical work on work-mugging also suggests the importance of preemption [3]. Even so, work-biasing can sometimes have a small benefit (approximately 1%) and never hurts performance, so we include work-biasing in our baseline work-stealing runtime.

We choose to include serial-sprinting and work-biasing in our baseline runtime to ensure that it is as aggressive as possible, but this does mean our baseline runtime includes a very limited form of asymmetry awareness.

IV. Evaluation Methodology

We use a vertically integrated research methodology spanning software, architecture, and VLSI. In this section, we describe our target system, applications, baseline runtime, cycle-level performance modeling, and energy modeling.

A. Target System

Although much of our analysis is applicable to larger high-performance systems, we focus on the smaller embedded systems that are already integrating static and dynamic asymmetry [17, 27–29]. Table I includes details on the core microarchitectures and memory system. We study two eight-core configurations: a 4B4L system with four big and four little cores similar to commercial products [28, 29] and a 1B7L system with one big core and seven little cores. We target a 32-bit RISC architecture with 32 general-purpose registers and hardware floating point. We specifically target an older technology (TSMC 65nm LP) and a lower-frequency, single-issue, in-order little core for two reasons. First, we have our own VLSI implementation of such a core and thus can pursue a more rigorous energy-modeling strategy (see Section IV-E).

TABLE I. CYCLE-LEVEL SYSTEM CONFIGURATION

Technology	TSMC 65nm LP, 1.0V nominal voltage
ALU	4/10-cycle Int Mul/Div, 6/6-cycle FP Mul/Div, 4/4-cycle FP Add/Sub
Little Core	1-way, 5-stage in-order, 32 Phys Regs, 333MHz nominal frequency
Big Core	4-way, out-of-order, 128 Phys Regs, 32 Entries IQ and LSQ, 96 Entries ROB, Tournament Branch Pred, 333MHz nominal frequency
Caches	1-cycle, 1-way, 16KB L1I, 1-cycle 2-way 16KB L1D per core; 20-cycle, 8-way, shared 1MB L2; MESI protocol
OCN	Crossbar topology, 2-cycle fixed latency
DRAM	200ns fixed latency, 12.8GB/s bandwidth SimpleMemory model

TABLE II. PERFORMANCE OF BASELINE RUNTIME VS. INTEL CILK++ AND INTEL TBB ON REAL SYSTEM

	Cilk++	**TBB**	**Baseline**	**Baseline vs. TBB**
dict	4.02	5.02	5.53	+10%
radix	7.05	4.87	5.58	+14%
rdups	3.96	4.36	4.54	+4%
mis	2.75	2.42	2.40	-1%
nbody	7.37	7.10	6.95	-3%

Numbers are speedups vs. scalar implementation. Cilk++ = original Cilk implementation of PBBS apps compiled with Intel C++ Compiler 14.0.2. TBB = ported PBBS apps using `parallel_for` with Intel TBB 4.4 build 20150928. Baseline = ported PBBS apps using `parallel_for` with our baseline work-stealing runtime. Each configuration uses eight threads running on an unloaded Linux server with two Intel Xeon E5620 processors.

Second, we have access to SPICE-level models of integrated voltage regulators in this technology that help us to accurately estimate DVFS transition overheads. We expect the high-level conclusions of our work to hold for high-performance target systems.

B. Benchmark Suite

We have ported 20 C++ application kernels to our RISC architecture and work-stealing runtime. These kernels are selected from the PBBS (v. 0.1) [57], PARSEC (v. 3.0) [5], Cilk (v. 5.4.6) [22], and UTS (v. 2.1) [53] benchmark suites and represent diverse application-level characteristics (see Table III). We include two datasets for *qsort* and *radix*, since they exhibit strong data-dependent variability. *bfs-d* and *bfs-nd* capture the impact of deterministic execution for the same problem. Determinism is a desirable trait that can mitigate the difficulties of reasoning about both correctness and performance in complex systems [57]. We list MPKI for each app, showing that our application kernels are fairly compute-bound. We select applications with varied parallelization methods. In addition to the conventional `parallel_for` construct, our selections exhibit recursive spawn-and-sync parallelization as well as nested loop parallelism (*sampsort* and *uts*). Most PBBS benchmarks parallelize execution with reserve-and-commit phases to create determinism. For *sptree* and *mis*, we choose the non-deterministic versions which use atomic memory operations to synchronize work. When running on our target systems, our application kernels achieve respectable parallel speedup and yet vary widely in the number of tasks and sizes of tasks. For detailed kernel descriptions, see [5, 22, 53, 57].

C. Work-Stealing Runtime

Work-stealing runtimes can be divided into several categories (e.g., language-based vs. library-based, child-stealing vs. continuation-stealing). While we believe that our approach can be applied to various categories, in this work we choose to focus on a C++ library-based implementation similar in spirit to Intel TBB. We support syntax similar to TBB's `parallel_for` and `parallel_invoke`. Our runtime uses child-stealing and supports automatic recursive decomposition of parallel loops (similar to Intel TBB's `simple_partitioner`). We use non-blocking, dynamically sized Chase-Lev task queues [12] and occupancy-based victim selection [15]. We have carefully optimized our runtime to minimize memory fences, atomic memory operations, and false sharing.

We have compared the performance of our baseline runtime to Intel Cilk++ and Intel TBB on five application kernels from PBBS running natively on an eight-core Intel x86 platform. We use large datasets and many trials so that it takes ≈30 seconds to run one serial application. The speedup results over optimized serial implementations are shown in Table II. Our runtime has similar performance to Intel TBB and is sometimes slightly faster due to the fact that our runtime is lighter weight and does not include advanced features like C++ exceptions or cancellations from within tasks. Section V uses cycle-level simulation to show that our baseline runtime achieves very reasonable speedups on both 4B4L and 1B7L systems. These real-system- and simulation-based results provide compelling evidence for the strength of our baseline runtime. As mentioned in Section III-C, we also add serial-sprinting and work-biasing (limited forms of asymmetry awareness) to our baseline runtime to ensure it is as aggressive as possible. Our AAWS runtime extends this baseline runtime as described in Sections III-A and III-B.

D. Cycle-Level Performance Modeling

We use the gem5 simulator [6] in syscall emulation mode for cycle-level performance modeling of our target systems. Heterogeneous systems are modeled by combining modified `O3CPU` and `InOrderCPU` models. We modified gem5 to toggle an activity bit in each core after executing the hint instructions. We modified gem5's clock domains and clocked object tick calculations to support dynamic frequency scaling. Cores can independently scale their frequency, but we centralized control of all clock domains in a DVFS controller specialized for AAWS. As described in Section III-A, we model a lookup-based DVFS controller that maps activity information into appropriate voltage levels. We use tables similar to those described in FG-SYNC+ [24]. We slightly modify this approach by separating little-core activity bits from big-core

TABLE III. APPLICATION KERNELS

									Speedup				
				DInst	Num	Task	Opt IO			1B7L		4B4L		L2
Name	Suite	Input	PM	(M)	Tasks	Size (K)	Cyc (M)	ERatio	O3	vs O3	vs IO	vs O3	vs IO	MPKI
bfs-d	pbbs	`randLocalGraph_J_5_150K`	p	36.0	2588	14	113.2	2.8	2.2	2.3	5.1	2.9	6.5	14.8
bfs-nd	pbbs	`randLocalGraph_J_5_150K`	p	58.1	3108	19	113.2	2.8	2.2	1.8	4.0	2.4	5.3	12.3
qsort-1	pbbs	`exptSeq_10K_double`	rss	18.8	777	24	26.1	2.5	1.7	2.8	4.7	3.2	5.4	0.0
qsort-2	pbbs	`trigramSeq_50K`	rss	20.0	3187	6	38.9	3.1	1.9	3.3	6.3	4.6	8.7	0.0
sampsort	pbbs	`exptSeq_10K_double`	np	37.5	15522	2	26.1	2.5	1.7	2.5	4.2	3.0	5.1	0.11
dict	pbbs	`exptSeq_1M_int`	p	45.1	256	151	101.5	2.8	1.7	4.0	6.9	5.1	8.8	7.0
hull	pbbs	`2Dkuzmin_100000`	rss	14.2	882	16	31.6	2.1	2.2	3.4	7.5	4.4	9.8	6.0
radix-1	pbbs	`randomSeq_400K_int`	p	42.4	176	240	83.1	2.2	1.8	2.7	4.7	3.1	5.5	7.7
radix-2	pbbs	`exptSeq_250K_int`	p	35.1	285	123	56.6	2.1	1.8	2.8	4.9	3.1	5.5	7.5
knn	pbbs	`2DinCube_5000`	p, rss	83.3	3499	23	139.3	2.8	1.7	6.0	9.9	7.0	11.5	0.02
mis	pbbs	`randLocalGraph_J_5_50000`	p	5.8	3230	2	11.6	3.6	2.3	3.8	9.0	4.3	10.1	3.5
nbody	pbbs	`3DinCube_180`	p, rss	56.6	485	116	75.1	2.9	1.6	5.6	8.7	7.1	11.1	0.01
rdups	pbbs	`trigramSeq_300K_pair_int`	p	51.2	288	156	108.4	2.6	1.7	3.5	5.9	4.2	7.1	7.6
sarray	pbbs	`trigramString_120K`	p	42.1	2434	16	114.7	2.5	2.3	2.6	6.0	2.9	6.8	10.0
sptree	pbbs	`randLocalGraph_E_5_100K`	p	18.9	482	39	57.2	2.8	2.1	3.0	6.3	3.5	7.3	4.9
clsky	cilk	`-n 128 -z 256`	rss	42.0	3645	11	70.4	2.4	1.7	5.1	8.6	6.2	10.5	0.02
cilksort	cilk	`-n 300000`	rss	47.0	2056	22	76.2	3.7	1.3	5.7	7.3	6.3	8.1	2.3
heat	cilk	`-g 1 -nx 256 -ny 64 -nt 1`	rss	54.3	765	54	64.9	2.3	2.1	4.2	8.8	5.7	11.7	0.04
ksack	cilk	`knapsack-small-1.input`	rss	30.1	78799	0.3	25.9	2.4	1.9	2.3	4.3	2.7	5.0	0.0
matmul	cilk	`200`	rss	68.2	2047	33	118.8	2.0	3.6	2.7	10.0	4.8	17.4	0.0
bscholes	parsec	`1024 options`	p	40.3	64	629	52.7	2.4	1.9	4.2	7.9	5.5	10.4	0.0
uts	uts	`-t 1 -a 2 -d 3 -b 6 -r 502`	np	63.9	1287	50	82.6	2.3	2.0	4.4	8.8	5.8	11.6	0.02

Suite = benchmark suite. Input = input dataset & options. PM = parallelization methods: p = `parallel_for`, np = nested `parallel_for`, rss = recursive spawn-and-sync. DInsts = dynamic instruction count in millions. Num Tasks = number of tasks. Task Size = average task size in thousands of instructions. Opt IO Cyc = number of cycles of an optimized *serial* implementation on an in-order core. ERatio = energy ratio of the serial implementation on O3 over IO (i.e., α in Section II-A). O3 = speedup of the serial implementation on O3 over IO (i.e., β in Section II-A). 1B7L = speedup on one big and seven little cores. 4B4L = speedup on four big and four little cores. L2 MPKI = L2 misses per one thousand instructions with the parallelized implementation and baseline runtime on one core.

activity bits. The number of active little cores and active big cores then maps to appropriate voltage levels according to the marginal utility model.

We use SPICE-level models of integrated voltage regulators in this technology to accurately estimate DVFS mode transition overheads. The transition time from 0.7 V to 1.33 V is roughly 160 ns with a sophisticated scheme as described in [24]. We model transition overheads linearly with 40 ns per 0.15 V step in gem5 to capture the general trend, and we include this overhead in our results. However, transitions happen infrequently with an average of 0.2 transitions per ten microseconds across our benchmarks (maximum of 0.7). We ran a sensitivity study sweeping transition overhead to 250 ns per step and saw less than 2% overall performance impact. Throughout our experiments, we assume that cores can continue executing through the voltage transition at the lower frequency, and we also assume that new decisions cannot be made until the previous transition completes.

We added support for the new `mug` instruction which can cause one core to initiate an interrupt on another core. Specific overheads are difficult to isolate but are modeled in our simulators: pipeline-flush overhead is captured through the gem5 interrupt-handling mechanism; register state (32 GPRs, exception cause and EPC, thread-ID reg) swapping is done via memory in the exception handler and captured through gem5 modeling of cache coherence and misses; instruction/-data cache migration overhead is captured the same way; we estimate the inter-core interrupt latency to be on the order of an L2 access and thus add an explicit 20-cycle latency per-mug. Because we use multithreaded workloads, we do not model TLB invalidations. Thread-swapping assembly includes about 80 instructions of mugging assembly code. We observe that work-mugging happens infrequently (less than 40 per million instructions) and that performance is generally insensitive to work-mugging overheads. We ran a sensitivity study sweeping the interrupt latency to 1000 cycles and saw less than 1% overall performance impact.

E. Energy Modeling

To help estimate energy of a little core, we developed a realistic RTL implementation of an in-order, single-issue scalar core and L1 memory system. The RTL model is synthesized and placed-and-routed using a combination of Synopsys Design Compiler, IC Compiler, and PrimeTime PX with a TSMC 65 nm LP standard-cell library characterized at 1 V. Analysis of the placed-and-routed design indicates each core is approximately 0.75 mm^2 and can run at 333 MHz at 1 V. We predict that more aggressive RTL and circuit design could increase this clock frequency by 2× or more. We then ran a suite of 65 *energy microbenchmarks* that are each designed to measure the energy used by various components in the core for each instruction. For example, the `addiu` energy

microbenchmark warms up the instruction cache (to isolate the energy solely due to the instruction under test) and then executes 100 `addiu` instructions in sequence. We ran this microbenchmark on the synthesized gate-level design to obtain bit-accurate traces that are fed into PrimeTime power simulations for power estimates. Coupled with the cycle time of the placed-and-routed design, we can calculate the energy per unit used by the `addiu` instruction.

We experimented with a purely McPAT-based energy model, but we had difficulty validating the McPAT in-order energy model against our own VLSI results. Since we do not have access to RTL for an aggressive superscalar out-of-order processor, we used a hybrid approach. We run the same energy microbenchmarks mentioned above on gem5's in-order model to generate event counts (e.g., register file reads, instruction cache accesses, integer ALU accesses). We then use our VLSI implementation to carefully calculate the energy for each event. We have developed a simple energy modeling tool which takes the event counts and our own component-level energy numbers and produces a total energy estimate. We iterate to ensure that the overall energy of each energy microbenchmark correlates between our VLSI implementation and our energy model. We then use McPAT's component-level models to estimate the energy of various structures within an out-of-order core but not within an in-order core. Since the absolute energy numbers from McPAT and our VLSI implementation do not necessarily match, we use the energy of a component that is present both in our VLSI implementation and McPAT (e.g., the integer ALU or register file read access) to normalize the McPAT component models. Finally, we account for pipeline register overhead and leakage power. We then run our energy microbenchmarks on our VLSI implementation, the gem5 in-order model, and the gem5 out-of-order model, and generate detailed energy breakdowns for every component. We carefully compare all breakdowns for each microbenchmark to ensure that our energy model matches our intuition.

SPICE-level simulations were used to determine the relationship between frequency and voltage for our cores across different operating modes. We used nine delay stages consisting of multiple FO4 loaded inverters, NAND, and NOR gates connected in a loop, such that the total delay in the loop matches our RTL cycle time for a given voltage. We used the change in delay vs. supply voltage as a model for core voltage-frequency scaling, and found the linear model described in Section II-A to be a good fit. We use the first-order model developed in Section II-A to estimate the energy as a function of DVFS scaling.

V. Evaluation Results

In this section, we evaluate the performance and energy efficiency of our AAWS runtime with *work-pacing*, *work-sprinting*, and *work-mugging* against our baseline system.

A. Performance of Baseline Work-Stealing Scheduler

Table III provides detailed performance and energy statistics for optimized serial code running on our single-core systems as well as for the parallelized versions running on our 1B7L and 4B4L systems. The big out-of-order core shows reasonable energy efficiency overhead and speedup compared to the little in-order core, similar to reported ratios collected by ARM during pre-silicon design-space exploration [25]. The 4B4L system strictly increases performance over the 1B7L system, although we observe that the additional big cores do not always provide much performance benefit. Figure 7(a) shows per-core activity of the baseline 4B4L system executing *radix-2*. Notice that the execution time of *radix-2* is limited by LP regions created by lagging little cores.

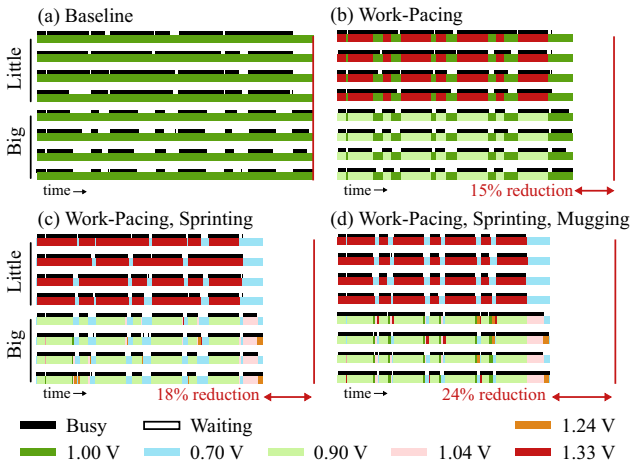


Figure 7. Activity Profiles for *radix-2* on 4B4L – Execution times of (b), (c), and (d) normalized to (a). Each row corresponds to a core's activity (black strip) and DVFS operating mode (colored strip) over time. (a) baseline 4B4L system; (b) applying work-pacing reduces HP region; (c) combining work-pacing and -sprinting reduces both HP and LP regions; (d) the complete AAWS runtime with work-pacing, sprinting, and mugging reduces execution time by 24%.

B. Performance Analysis of Work-Pacing, Work-Sprinting, and Work-Mugging

In this subsection, we evaluate the performance benefit of work-pacing, work-sprinting, and work-mugging on both target systems. Figure 8 shows detailed execution time breakdowns for the (a) 4B4L system and the (b) 1B7L system. Each group of bars represents a single application, and bars incrementally add our techniques. To aid our evaluation, breakdowns within each bar represent the time spent in the serial region (*serial*), the HP region (*HP*), and the LP region. The LP region is further broken down into three categories. First, we isolate the LP region within which the number of inactive big cores is too few to mug all work. In this region, the number of *big inactive* cores is fewer than the number of *little active* cores (*BI<LA*). Second, we isolate the LP region within which inactive big cores can mug all work from little cores. In this region, the number of *big inactive* cores matches or exceeds the number of *little active* cores (*BI>=LA*). Lastly, we gather the remaining LP region in which mugging is not possible into a single "other LP" category (*oLP*). The system with all of our techniques together is represented by *base+psm*. Note that the bar with work-mugging alone (*base+m*) serves as a comparison point without marginal utility techniques.

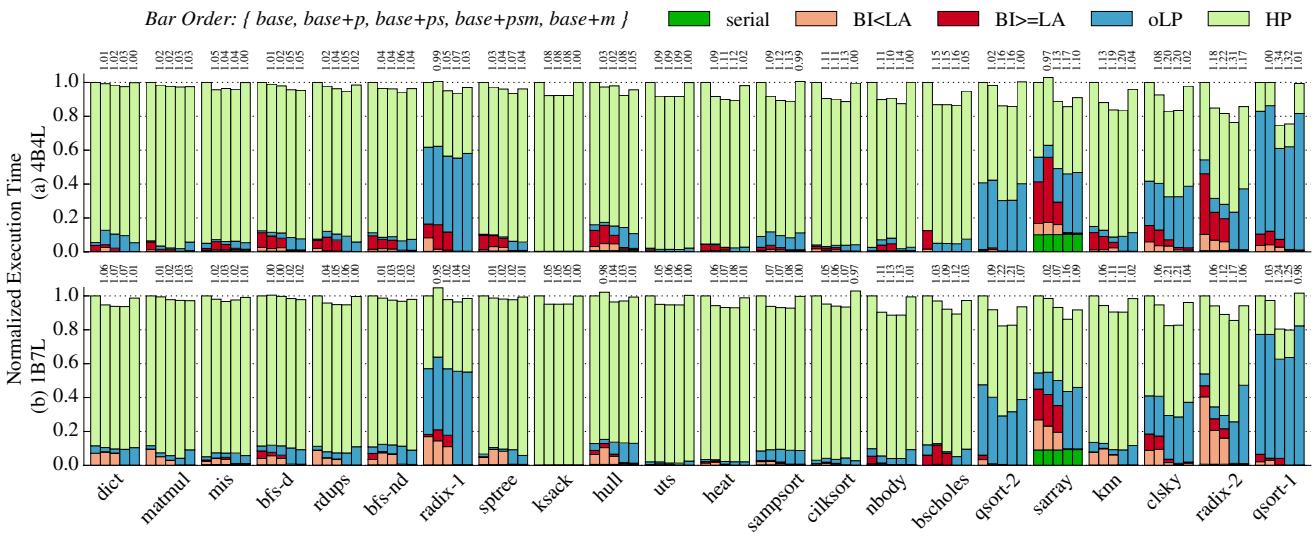


Figure 8. Normalized Execution Time Breakdown – (a) 1B7L configuration; (b) 4B4L configuration. Each group of bars represents a single application. First bar within each group is baseline with work-biasing and serial-region sprinting (*base*). Additional bars incrementally add our techniques: work-pacing (*base+p*); work-pacing and work-sprinting (*base+ps*); work-pacing, work-sprinting, and work-mugging (*base+psm*); work-mugging alone (*base+m*). All execution times are normalized to the baseline (*base*), and speedup is printed above each bar. Kernels are sorted by speedup for 4B4L *base+psm* from left to right. See Section V for breakdown explanations.

We evaluate work-pacing by comparing *base+p* to *base*. Work-pacing achieves reasonable performance benefits despite all cores often being completely busy (i.e., in the HP region) simply by applying a marginal-utility-based approach to the core voltages and frequencies. However, optimizing instantaneous system throughput can either mitigate or exacerbate load imbalance with realistic task sizes. For example, in the execution of *sarray* (4B4L), *radix-1* (1B7L and 4B4L), and *hull* (1B7L), benefits are outweighed by newly created lagging threads. Similarly, *qsort-1* on 4B4L sees smaller benefit because although the small HP region is sped up, the critical big core that generates new work is slowed down. Load balancing can also sometimes *improve* when lagging threads are coincidentally eliminated, thereby reducing the LP region (e.g., *radix-2* 1B7L and 4B4L, *qsort-2* 1B7L). With $\alpha \approx 3$ and $\beta \approx 2$ (see Table III), our analytical modeling suggests up to 12% benefit in the HP region (see Figure 4). Our results suggest that although work-pacing can sometimes achieve these predicted speedups, newly lagging threads can also limit the potential benefit. We can significantly mitigate this effect by applying work-sprinting in the LP region.

We evaluate work-sprinting by comparing *base+ps* to *base+p*. Work-sprinting rests waiting cores for power slack and then applies a marginal-utility-based approach to tune the voltage and frequency of active cores within the LP region. Notice that applications with large LP regions (i.e., combination of *BI<LA*, *BI>=LA*, and *oLP* regions) have the greatest performance improvements (e.g., *qsort-1, radix-2, clsky, knn, sarray, qsort-2*). Negative latency impacts from work-pacing are also significantly reduced (e.g., *sarray* 4B4L, *qsort-1* 4B4L, *radix-1* 1B7L and 4B4L). Note that *radix-1* sees little benefit because all four big cores are busy in the LP region, and the resting little cores do not generate enough power slack to sprint. Datasets can significantly impact the size of LP regions. In particular, note that *qsort-1* sorts an exponentially distributed dataset, making tasks very short or very long and creating large LP regions that work-sprinting can then exploit. Although work-sprinting allows little cores to sprint lagging threads, it is often optimal to move the work onto big cores and sprint the big cores when possible (i.e., *BI>=LA* regions). We add the final AAWS technique, work-mugging, to take advantage of these opportunities for further benefit.

We evaluate work-mugging by comparing *base+psm* to *base+ps* as well as *base+m* to *base*. Work-mugging helps the LP region by moving work from slow little cores to fast big cores. First, notice that work-mugging eliminates all *BI<LA* and *BI>=LA* regions (i.e., all opportunities for work mugging are exhausted). In general, the movement of work between cores can result in smaller speedups (e.g., *bscholes* 1B7L, *sptree* 4B4L) as well as larger speedups (e.g., *hull* 4B4L, *radix-2* 4B4L, *sarray* 1B7L). Work-mugging provides the greatest benefit on applications with large *BI>=LA* regions, in which all work can be mugged onto fast big cores. Performance benefit is more limited in the *BI<LA* region in which some work must be left executing slowly on little cores. Although rarely the case, mugging overheads can cause minor slowdown. For example, *qsort-1* on 1B7L experiences slight slowdown with *base+m* compared to *base* when many tiny tasks at the end of the sort are quickly spawned and successively mugged onto the big core, incurring mugging overhead each time.

Figures 7(b-d) show activity profiles for *radix-2* executing on the 4B4L system. By comparing Figure 7(a) and (b) we can see that during the HP region, the AAWS runtime tunes performance by raising the voltage of little cores and lowering the voltage of big cores. In Figure 7(c), we see that the AAWS runtime rests waiting cores and then sprints the remaining active cores. Most of the leftover work in the LP region is executing on little cores, limiting the performance benefit from

work-sprinting. The little cores quickly reach their maximum voltage and frequency and leave a large amount of power headroom unused. Finally, Figure 7(d) shows work-mugging shifting tasks from little cores to big cores in the LP region for a total execution time reduction of 24%.

In summary, work-pacing can provide performance benefit in the HP region as suggested by our analytical modeling, but due to realistic task sizes, the overall performance can vary widely when applied alone. The benefits of work-sprinting and work-mugging depend heavily on the presence of LP regions, which can vary with the application, the algorithm, and the dataset. Either technique can be applied independently to improve performance in the LP region. However, when non-ideal work-stealing schedules result in the presence of *BI>=LA* regions, work-sprinting and work-mugging can complement each other to provide the largest speedups. Finally, work-mugging and DVFS transition overheads have minor performance impacts, likely due to the relative infrequency of mugs and DVFS transitions.

C. Performance Versus Energy Analysis

Figure 9 shows detailed performance and energy efficiency results for the 4B4L system. Notice that the general trend is higher performance and higher energy efficiency at similar power (i.e., tracking the isopower line). A more sophisticated adaptive DVFS control scheme with performance and energy counters could track the isopower line more closely, but a simple lookup-table-based DVFS controller is less complex and can generally improve both performance and energy efficiency with slight power variation. Figure 9(a) compares *base*, *base+p*, and *base+ps* and shows that work-pacing alone can improve both performance and energy efficiency for many applications at similar power. Several applications suffer reduced performance and energy efficiency due to the creation of lagging threads. Work-sprinting increases performance and energy efficiency, mitigating the impact of lagging threads. Detailed energy breakdown data (not shown) suggests that work-pacing and work-sprinting save energy because: (1) big cores execute at low-voltage operating modes, and (2) slower big cores do less work, allowing work to be stolen and executed on more energy-efficient little cores.

Figure 9(b) compares *base*, *base+psm*, and *base+m*. We show results for *base+m* as a comparison point without marginal utility techniques. Detailed energy breakdown data suggests that work-mugging significantly reduces the busy-waiting energy of cores in the steal loop, which are operating at nominal voltage and frequency without work-sprinting. We therefore notice that *base+m* improves both performance and energy efficiency. The complete AAWS runtime (*base+psm*) provides the greatest improvements across all applications. The strengths and weaknesses of work-pacing, work-sprinting, and work-mugging complement each other to elegantly adapt to diverse application-level behavior.

VI. Related Work

While the MIT Cilk project helped recently popularize work-stealing [7, 8, 22], the general idea dates to at least the

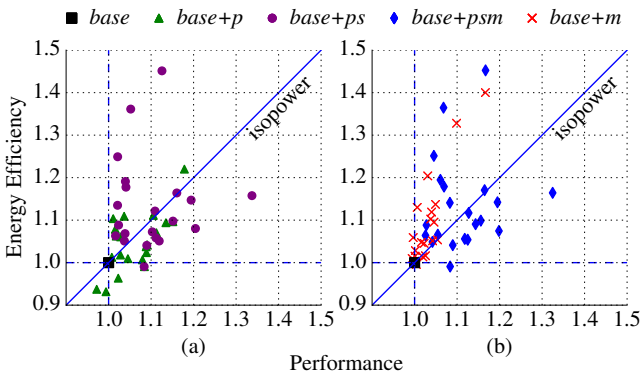


Figure 9. Energy Efficiency vs. Performance – Each point represents one application kernel's performance and energy efficiency running with a particular subset of the AAWS techniques, normalized to the same application kernel running on the baseline 4B4L system. Black markers on the origin represent *base* in both plots. An isopower line is drawn for reference, where points below the line are higher power and points above the line are lower power compared to the baseline. (a) *base*, *base+p*, *base+ps*. (b) *base*, *base+psm*, *base+m*.

early 1980's [10, 30]. There has been tremendous work over the past decade on work-stealing runtimes (e.g., optimized task queue implementations [12], alternative victim selection strategies [4, 15], efficiently supporting reduction operations across tasks [47]). However, very little work explores the interaction between work-stealing schedulers and either static or dynamic asymmetry with a few notable exceptions.

Bender and Rabin proposed work-mugging as a way to migrate work from slow to fast cores and analyzed the theoretical impact of work-mugging on performance [3]. Follow-up work by Jovanović and Bender used high-level discrete-event simulation to explore the potential benefits and overheads of work-mugging [37]. We build on this earlier work with a realistic implementation and a new context. Chronaki et al. propose a dynamic task scheduler with constrained work stealing that attempts to schedule critical tasks to big cores [14]. This technique is most effective in applications with low parallel slack. Costero et al. [16] group a big and little core together into a virtual core for the work-stealing runtime, and use a completely separate customized scheduler within the virtual core. Chen et al. propose the workload-aware task scheduler (WATS) which uses a combination of history-based task-execution-time prediction and task affinity [13]. WATS demonstrates good performance but is relatively complex and uses randomized victim selection in the baseline work-stealing runtimes. Previous work has shown the benefit of occupancy-based victim selection [15]. Ribic et al. proposed a work-stealing runtime that exploits dynamic asymmetry to improve energy efficiency, and they report reasonable energy benefits with modest performance loss on a real system [55]. Our proposal is fundamentally different in that it focuses on improving both performance and energy efficiency by exploiting both static and dynamic asymmetry.

There has been a wealth of research on scheduling for statically asymmetric systems [1, 35, 36, 42–44, 52, 59, 60]. Most closely related to our work are techniques that acceler-

ate applications written with a thread-based parallel programming framework [16, 35, 36, 44, 52, 59]. For example, Joao et al. propose bottleneck identification and scheduling which migrates programmer-identified bottlenecks to big cores [35], Lakshminarayana et al. propose progress performance counters to accelerate lagging threads [44], and Joao et al. propose utility-based acceleration to accelerate both lagging and bottleneck threads [36]. These prior works focus on traditional thread-based parallel programming frameworks as opposed to task-based frameworks based on state-of-the-art work-stealing runtimes. They do not explore the interaction between static and dynamic asymmetry.

DVFS is perhaps one of the most well-studied techniques for power management [4, 9, 11, 19, 20, 24, 32–34, 40, 48, 51, 58]. Most closely related to our own work are techniques that accelerate multithreaded applications. For example, Miller et al. and Godycki et al. both propose instrumenting a traditional thread library to reactively sprint lagging threads in LP regions [24, 51]. Cai et al. and Bhattacharjee et al. use instrumentation or prediction to rest waiting threads and sprint lagging threads [4, 11]. While there are certainly similarities between this prior work and AAWS, there are unique opportunities involved in exploiting static and dynamic asymmetry within the context of a state-of-the-art work-stealing runtime.

Finally, some important work studies the tradeoffs between static and dynamic asymmetry, albeit with multiprogrammed workloads [2, 26, 50]. Azizi et al. use a similar marginal-utility-based approach to explore circuit/architecture co-design and the impact of voltage scaling, but their work is purely within the context of VLSI design as opposed to adaptive scheduling for runtime systems [2]. Lukefahr et al. argue that heterogeneous microarchitectures trump DVFS, but the study is within the context of a novel reconfigurable core (as opposed to static asymmetry) and uses multiprogrammed workloads scheduled optimally offline [50]. A key conclusion in these works is that heterogeneous microarchitectures can offer steeper utility curves, while DVFS offers a shallower tradeoff. We see the same tradeoff in Figure 3(a), and we exploit this in our marginal-utility-based approach.

VII. Conclusion

To our knowledge, this is the first work to explore the interaction between static asymmetry (in the form of heterogeneous microarchitectures), dynamic asymmetry (in the form of fine-grain DVFS), and work-stealing runtimes. We argue that work-stealing runtimes are a natural fit for managing asymmetry, but we also argue that there are unique opportunities for an asymmetry-aware work-stealing runtime. Through a mix of first-order modeling, numerical analysis, runtime software development, architecture-level simulation, and VLSI energy modeling, we have attempted to make the case that holistically combining static asymmetry, dynamic asymmetry, and work-stealing runtimes can improve performance and energy efficiency in future multicore systems.

Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, AFOSR YIP Award #FA9550-15-1-0194, and donations from Intel and Synopsys. The authors would like to thank Angelina Lee for her early feedback, and Alyssa Apsel, Ivan Bukreyev, and Waclaw Godycki for their help on circuit-level modeling for integrated voltage regulation.

References

[1] A. Annamalai et al. An Opportunistic Prediction-Based Thread Scheduling to Maximize Throughput/Watt in AMPs. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep 2013.

[2] O. Azizi et al. Energy-performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. *Int'l Symp. on Computer Architecture*, Jun 2010.

[3] M. A. Bender and M. O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems*, 35(3):289–304, Jun 2002.

[4] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. *Int'l Symp. on Computer Architecture*, Jun 2009.

[5] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Oct 2008.

[6] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug 2011.

[7] R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug 1996.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, Sep 1999.

[9] T. Burd et al. A Dynamic Voltage Scaled Microprocessor System. *IEEE Journal of Solid-State Circuits*, 35(11):1571–1580, Nov 2000.

[10] F. W. Burton and M. R. Sleep. Executing Functional Programs on a Virtual Tree of Processors. *Conf. on Functional Programming Languages and Computer Architecture*, Aug 1984.

[11] Q. Cai et al. Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Oct 2008.

[12] D. Chase and Y. Lev. Dynamic Circular Work-Stealing Deque. *Symp. on Parallel Algorithms and Architectures*, Jun 2005.

[13] Q. Chen et al. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures. *Int'l Parallel and Distributed Processing Symp.*, May 2012.

[14] K. Chronaki et al. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. *Int'l Symp. on Supercomputing*, Jun 2015.

[15] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. *Int'l Symp. on Workload Characterization*, Sep 2008.

[16] L. Costero et al. Revisiting Conventional Task Schedulers to Exploit Asymmetry in ARM big.LITTLE Architectures for Dense Linear Algebra. *CoRR arXiv:1509.02058*, Sep 2015.

[17] M. Demler. MediaTek Steps Up to Tablets: MT8135 Brings Heterogeneous Multiprocessing to Big.Little. *Microprocessor Report, The Linley Group*, Aug 2013.

[18] J. Dinan et al. Scalable Work Stealing. *Int'l Conf. on High Performance Networking and Computing*, Nov 2009.

[19] R. G. Dreslinski. *Near-Threshold Computing: From Single-Core to Many-Core Energy-Efficient Architectures*. Ph.D. Thesis, EECS Department, University of Michigan, 2011.

[20] S. Eyerman and L. Eechkout. Fine-Grained DVFS Using On-Chip Regulators. *ACM Trans. on Architecture and Code Optimization*, 8(1):1:1–1:24, Apr 2011.

[21] E. Fluhr et al. The 12-Core POWER8 Processor With 7.6 Tb/s IO Bandwidth, Integrated Voltage Regulation, and Resonant Clocking. *IEEE Journal of Solid-State Circuits*, 50(1):10–23, Jan 2015.

[22] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun 1998.

[23] H. Ghasemi et al. Cost-Effective Power Delivery to Support Per-Core Voltage Domains for Power-Constrained Processors. *Design Automation Conf.*, Jun 2012.

[24] W. Godycki et al. Enabling Realistic Fine-Grain Voltage Scaling with Reconfigurable Power Distribution Networks. *Int'l Symp. on Microarchitecture*, Dec 2014.

[25] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *EE Times*, Oct 2011.

[26] E. Grochowski et al. Best of Both Latency and Throughput. *Int'l Conf. on Computer Design*, Oct 2004.

[27] L. Gwennap. Renesas Mobile Goes Big (and Little). *Microprocessor Report, The Linley Group*, Feb 2013.

[28] L. Gwennap. Qualcomm Tips Cortex-A57 Plans: Snapdragon 810 Combines Eight 64-Bit CPUs, LTE Baseband. *Microprocessor Report, The Linley Group*, Apr 2014.

[29] L. Gwennap. Samsung First with 20 nm Processor. *Microprocessor Report, The Linley Group*, Sep 2014.

[30] R. H. Halstead. Implementation of Multilisp: Lisp on a Multi-Processor. *Symp. on Lisp and Functional Programming*, Oct 1984.

[31] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, Jul 2008.

[32] C. Isci et al. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. *Int'l Symp. on Microarchitecture*, Dec 2006.

[33] C. Isci and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. *Int'l Symp. on Microarchitecture*, Dec 2006.

[34] R. Jevtic et al. Per-Core DVFS with Switched-Capacitor Converters for Energy Efficiency in Manycore Processors. *IEEE Trans. on Very Large-Scale Integration Systems*, 23(4):723–730, Apr 2015.

[35] J. A. Joao et al. Bottleneck Identification and Scheduling in Multi-Threaded Applications. *Int'l Conf. on Architectural Support for Programming Languages and Operating Sys.*, Mar 2012.

[36] J. A. Joao et al. Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. *Int'l Symp. on Computer Architecture*, Jun 2013.

[37] N. Jovanović and M. A. Bender. Task Scheduling in Distributed Systems by Work Stealing and Mugging – A Simulation Study. *Int'l Conf. on Information Technology Interfaces*, Jun 2002.

[38] D. Kanter. Haswell's FIVR Extends Battery Life. *Microprocessor Report, The Linley Group*, Jun 2013.

[39] W. Kim, D. Brooks, and G.-Y. Wei. A Fully-Integrated 3-Level DC-DC Converter for Nanosecond-Scale DVFS. *IEEE Journal of Solid-State Circuits*, 47(1):206–219, Jan 2012.

[40] W. Kim et al. System-Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2008.

[41] K. Krewell. ARM Pairs Cortex-A7 With A15: Big.Little Combines A5-Like Efficiency With A15 Capability. *Microprocessor Report, The Linley Group*, Nov 2011.

[42] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Int'l Symp. on Microarchitecture*, Dec 2003.

[43] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Int'l Symp. on Computer Architecture*, Jun 2004.

[44] N. Lakshminarayana, J. Lee, and H. Kim. Age-Based Scheduling for Asymmetric Multiprocessors. *Int'l Conf. on High Performance Networking and Computing*, Nov 2009.

[45] H.-P. Le et al. A Sub-ns Response Fully Integrated Battery-Connected Switched-Capacitor Voltage Regulator Delivering 0.19 W/mm^2 at 73% Efficiency. *Int'l Solid-State Circuits Conf.*, Feb 2013.

[46] H.-P. Le, S. R. Sanders, and E. Alon. Design Techniques for Fully Integrated Switched-Capacitor DC-DC Converters. *IEEE Journal of Solid-State Circuits*, 46(9):2120–2131, Sep 2011.

[47] I.-T. A. Lee, A. Shafi, and C. E. Leiserson. Memory-Mapping Support for Reducer Hyperobjects. *Symp. on Parallel Algorithms and Architectures*, Jun 2012.

[48] J. Li, J. F. Martinez, and M. C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2004.

[49] S. Li et al. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Trans. on Architecture and Code Optimization*, 10(1):5:1–5:29, Apr 2013.

[50] A. Lukefahr et al. Heterogeneous Microarchitectures Trump Voltage Scaling for Low-Power Cores. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Aug 2014.

[51] T. N. Miller et al. Booster: Reactive Core Acceleration For Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2012.

[52] T. Y. Morad et al. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *Computer Architecture Letters*, 5(1), Jan 2006.

[53] S. Olivier et al. UTS: An Unbalanced Tree Search Benchmark. *Int'l Workshop on Lanaguages and Compilers for Parallel Computing*, Nov 2006.

[54] J. Park et al. Accurate Modeling and Calculation of Delay and Energy Overheads of Dynamic Voltage Scaling in Modern High-Performance Microprocessors. *Int'l Symp. on Low-Power Electronics and Design*, Aug 2010.

[55] H. Ribic and Y. D. Liu. Energy-Efficient Work-Stealing Language Runtimes. *Int'l Conf. on Architectural Support for Programming Languages and Operating Sys.*, Mar 2014.

[56] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Sys.*, Mar 2010.

[57] J. Shun et al. Brief Announcement: The Problem Based Benchmark Suite. *Symp. on Parallel Algorithms and Architectures*, Jun 2012.

[58] A. A. Sinkar et al. Low-Cost Per-Core Voltage Domain Support for Power-Constrained High-Performance Processors. *IEEE Trans. on Very Large-Scale Integration Systems*, 22(4):747–758, Apr 2014.

[59] M. A. Suleman et al. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. *Int'l Conf. on Architectural Support for Programming Languages and Operating Sys.*, Mar 2009.

[60] K. Van Craeynest et al. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). *Int'l Symp. on Computer Architecture*, Jun 2012.

[61] T. von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. *Int'l Symp. on Computer Architecture*, May 1992.

[62] X. Wang and J. F. Martinez. XChange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2015.

[63] D. H. Woo and H.-H. S. Lee. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *IEEE Computer*, 41(12):24–31, Dec 2008.

[64] G. Yan et al. AgileRegulator: A Hybrid Voltage Regulator Scheme Redeeming Dark Silicon for Power Efficiency in a Multicore Architecture. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2012.