Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips*

Timothy N. Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, Radu Teodorescu
Department of Computer Science and Engineering
The Ohio State University
{millerti, panxi, thomasr, sedaghat, teodores}@cse.ohio-state.edu
http://arch.cse.ohio-state.edu

Abstract

Lowering supply voltage is one of the most effective techniques for reducing microprocessor power consumption. Unfortunately, at low voltages, chips are very sensitive to process variation, which can lead to large differences in the maximum frequency achieved by individual cores. This paper presents Booster*, a simple, low-overhead framework for dynamically rebalancing performance heterogeneity caused by process variation and application imbalance. The* Booster *CMP includes two power supply rails set at two very low but different voltages. Each core can be dynamically assigned to either of the two rails using a gating circuit. This allows cores to quickly switch between two different frequencies. An on-chip governor controls the timing of the switching and the time spent on each rail. The governor manages a "boost budget" that dictates how many cores can be sped up (depending on the power constraints) at any given time. We present two implementations of* Booster*:* Booster VAR*, which virtually eliminates the effects of core-to-core frequency variation in near-threshold CMPs, and* Booster SYNC*, which additionally reduces the effects of imbalance in multithreaded applications. Evaluation using PARSEC and SPLASH2 benchmarks running on a simulated 32-core system shows an average performance improvement of 11% for* Booster VAR *and 23% for* Booster SYNC*.*

1. Introduction

Current industry trends point to a future in which chip multiprocessors (CMPs) will scale to hundreds of cores. Unfortunately, hard limits on power consumption are threatening to limit the performance of future chips. Today's high-end microprocessors are already reaching their thermal design limits [15, 28] and have to scale down frequency under high utilization. The International Technology Roadmap for Semiconductors (ITRS) [16] has recognized for a while that power reduction in future technology generations will become increasingly difficult. If current integration trends continue, chips could see a 10-fold increase in power density by the time 11nm technology is in production. This will not only limit chip frequency but will also restrict the number of cores that can be powered on simultaneously [37]. The only way to ensure continued scaling and performance growth is to develop solutions that dramatically increase computational energy efficiency.

A very effective approach to improving the energy efficiency of a microprocessor is to lower its supply voltage (V_{dd}) to very close to the transistor's threshold voltage (V_{th}), into the so-called near-threshold (NT) region [5, 8, 26, 29]. This is significantly lower than what is used in standard dynamic voltage and frequency scaling (DVFS), resulting in aggressive reductions in power consumption (up to 100×) with about a 10× loss in maximum frequency. The very low power consumption allows many more cores to be powered on than in a CMP at nominal V_{dd} (albeit at much lower frequency). Even with the lower frequency, CMPs running in near-threshold can achieve significant improvements in energy efficiency, especially for highly parallel workloads. A recent prototype of a low-voltage chip from Intel Corp. is showing very promising results [38].

Unfortunately, near-threshold chips are very sensitive to *process variation*. Variation is caused by extreme challenges with manufacturing chips with very small feature sizes. Variation affects crucial transistor parameters such as threshold voltage (V_{th}) and effective gate length (L_{eff}) leading to heterogeneity in transistor delay and power consumption. In a large CMP, variation can lead to large differences in the maximum frequency achieved by individual cores [14, 36]. Low-voltage operation greatly exacerbates these effects because of the much smaller gap between V_{dd} and V_{th}. For 22nm technology, variation at near-threshold

*This work was supported in part by the National Science Foundation under grant CCF-1117799 and an allocation of computing time from the Ohio Supercomputer Center.

978-1-4673-0826-7/12/$26.00 ©2011 IEEE

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

voltages can easily increase by an order of magnitude or more compared to nominal voltage [30].

One solution for dealing with frequency variation is to constrain the CMP to run at the frequency of the slowest core. This eliminates performance heterogeneity but also severely lowers performance, especially when frequency variation is very high [30]. Moreover, power is wasted on the faster cores, because they could achieve the same performance at a lower voltage. Another option is to allow each core to run at the maximum frequency it can achieve, essentially turning a CMP that is homogeneous by design into a CMP with heterogenous and unpredictable performance. Previous work has used thread scheduling and other approaches that exploit workload imbalance [13, 31, 35, 36] to reduce the impact of heterogeneity on CMP performance. These techniques are effective for single-threaded applications or multiprogrammed workloads. However, they still suffer from unpredictable performance when processor heterogeneity is variation-induced. Moreover, these techniques are less effective when applied to multithreaded applications.

This paper presents *Booster*, a simple, low-overhead framework for dynamically re-balancing performance heterogeneity caused by process variation or application imbalance. The *Booster* CMP includes two power supply rails set at two very low but different voltages. Each core in the CMP can be dynamically assigned to either of the two power rails using a gating circuit [17]. This allows each core to rapidly switch between two different maximum frequencies. An on-chip governor determines when individual cores are switched from one rail to the other and how much time they spend on each rail. A "boost budget" restricts how many cores can be assigned to the high voltage rail at the same time, subject to power constraints.

We present two implementations of *Booster*: *Booster VAR*, which virtually eliminates the effects of core-to-core frequency variation, and *Booster SYNC*, which reduces the effects of imbalance in multithreaded applications.

With *Booster VAR* the governor maintains an average per-core frequency that is the same across all cores in the CMP. To achieve this, the governor schedules cores that are inherently slow to spend more time on the high voltage rail while those that are fast will spend more time on the low voltage rail. A schedule is chosen such that frequencies average to the same value over a finite interval. The result is a CMP that achieves performance homogeneity much more efficiently than is possible with a single supply voltage.

The goal of *Booster SYNC* is to reduce the effects of workload imbalance that exists in many multithreaded applications. This imbalance is caused by application characteristics, such as uneven distribution of work between threads, or runtime events like cache misses, which can cause non-uniform delays. Unbalanced applications lead to inefficient resource utilization because fast threads end up idling at synchronization points, waisting power [1, 23]. *Booster SYNC* addresses this imbalance with a voltage rail assignment schedule that favors cores running high-priority threads. These cores are given more time on the high-voltage rail at the expense of the cores running low-priority threads. *Booster SYNC* uses hints provided by synchronization libraries to determine which cores should be boosted. Unlike in previous work that addressed this problem [1, 23], the goal is not to save power by slowing down non-critical threads but to improve performance by reducing workload imbalance.

Evaluation of the *Booster* system on SPLASH2 and PARSEC benchmarks running on a simulated 32-core system shows that *Booster VAR* reduces execution time by 11%, on average, over a baseline heterogeneous CMP with the same average frequency. Compared to the same baseline, *Booster SYNC* reduces runtime by 19% and reduces the energy delay product by 23%.

This paper makes the following main contributions:

- The first solution for virtually eliminating core-to-core frequency variation in low-voltage CMPs.
- A novel solution for speeding up unbalanced parallel workloads.
- A hardware mechanism that uses synchronization library hints to track thread and core relative priority.

This paper is organized as follows: Section 2 presents the proposed *Booster* framework. Sections 3 and 4 describe the *Booster VAR* and *Booster SYNC* implementations. Sections 5 and 6 discuss the methodology and results of our evaluation. Section 7 discusses related work and Section 8 concludes.

2. The Booster Framework

The *Booster* framework relies on the CMP's ability to frequently change the voltage and frequency of individual cores. To ensure reliable operation, execution must be stopped while the voltage is in transition and the clock locks on the new frequency. To keep the performance overhead low, this transition must be very fast. Standard DVFS is generally driven by off-chip voltage regulators, which react slowly, requiring dozens of microseconds per transition. On-chip regulators could allow faster switching and potentially core-level DVFS control and have shown promising results in prototypes [18]. They are, however, costly to implement since one regulator per core is required if core-level control is needed. They also suffer from low efficiency because they run at much higher frequencies than their off-chip counterparts. Even the fastest on-chip regulators require hundreds to thousands of cycles to change voltage [18, 19].

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

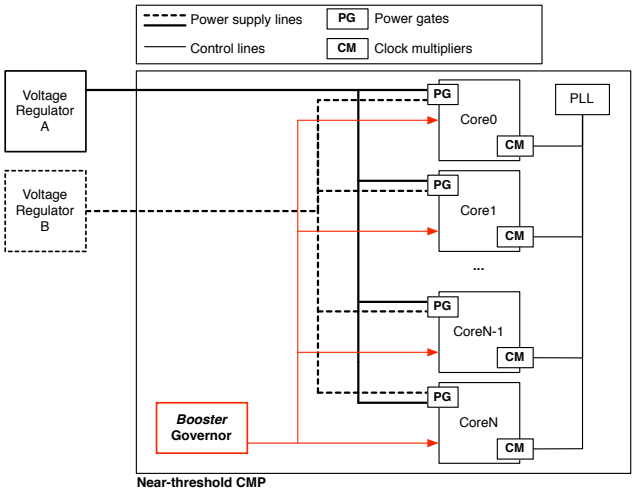

Figure 1. Overview of the Booster framework.

2.1. Core-Level Fast Voltage Switching

We use a different approach to control voltage and frequency levels at core granularity. In the *Booster* framework all cores are supplied with two power rails set at two different voltages. At near-threshold even small changes in V_{dd} have a significant effect on frequency. Thus, even a small difference (100-200mV) between the two rails gives cores a significant frequency boost (400-800MHz). Two external voltage regulators are required to independently regulate power supply to the two rails as shown in Figure 1. To keep the overhead of the additional regulator low, the sizes of the off-chip capacitors can be reduced significantly because each regulator handles a smaller current load in the new design. Each core in the CMP can be dynamically assigned to either of the two power rails using gating circuits [17, 22] that allow very fast transition between the two voltage levels. Within each core, only a single power distribution network is needed, leaving the core layout unchanged.

To measure how quickly *Booster* can change voltage rails, we conducted SPICE simulations of a circuit that uses RLC blocks to represent the resistance, capacitance and inductance of processor cores. The simulated circuit is shown in Figure 2(a). The RLC data represents Nehalem processors and is taken from [22]. This simple RLC model does not capture all effects of the voltage switch on the power distribution network, but it offers a good estimate of the voltage transition time. We simulate the transition of a single core between two voltage lines: low V_{dd} at 400mV and high V_{dd} at 600mV. A load equivalent to 15 cores is on the high V_{dd} line and one equivalent to 15 cores is on the low V_{dd} line at the time of the transition. Two power gates (M1 and M2), implemented with large PMOS transistors, are used to connect the test core to either the 600mV or the 400mV line. The gates were sized to handle the maximum

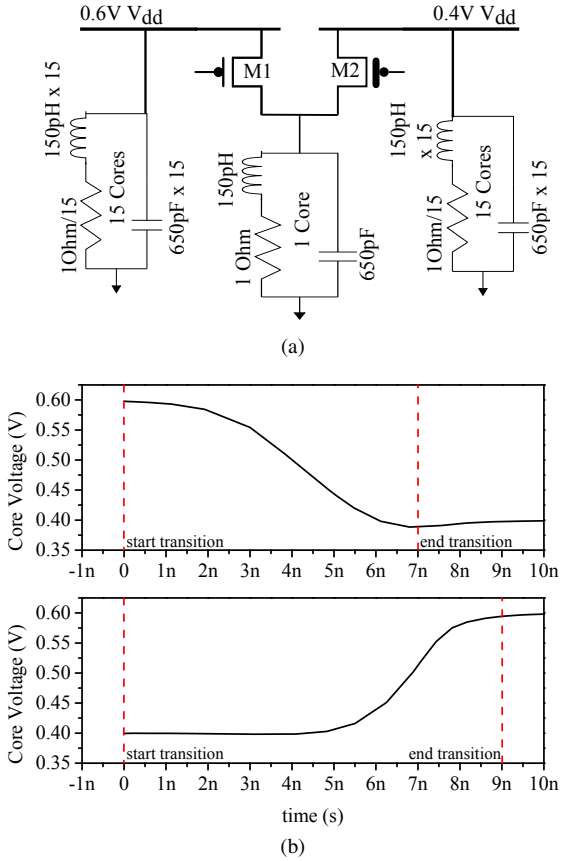

Figure 2. (a) Diagram of circuit used to test the speed of power rail switching for 1 core in a 32 core CMP. (b) Voltage response to switching power gates; control input transition starts at time=0.

current that can be drawn by each core. Both transistors were sized to have very low on-channel resistance (1.8 milliohms) to minimize the voltage drop across them.

Figure 2(b) shows the V_{dd} change at the input of the core in transition, when the core switches from high voltage to low (top graph) and from low voltage to high (bottom graph). During a transition the core is clock-gated to ensure reliable operation. As the graphs show, the transition from 600mV to 400mV takes about 7ns. Switching from 400mV to 600mV takes closer to 9ns, which is 9 cycles at 1GHz, the average frequency at which the *Booster* CMP runs. In our experiments we conservatively model a 10 cycle transition time. A similar voltage change takes tens of microseconds if performed by an external voltage regulator.

This experiment shows that changing power rails adds very little time overhead even if performed frequently. Power gates do introduce an area overhead to the CMP design. Per core, two gates have an area equivalent to about 6K transistors. For 32 cores this adds an overhead of ~192K transistors, or less than 0.02% of a billion transistor chip.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

2.2. Core-Level Fast Frequency Switching

Booster also requires core-level control over frequency. We assume a clock distribution and regulation system similar to the one used in the Intel Nehalem family [21]. Nehalem uses a central PLL to supply multiple phase-aligned reference frequencies, and distributed PLLs generate the clock signals for each core. This design allows core frequencies to be changed very quickly with 1-2 cycles of overhead when the clock has to be stopped. *Booster* requires a larger number of discrete frequencies than Nehalem because it allows each core to run at its maximum frequency (in steps of 25MHz in our implementation). In order to obtain a larger number of discrete frequencies, a reference signal generated by a central PLL is supplied to each core. Each core uses a clock multiplier [27, 33], which generates multiples of the base frequency. These multipliers have been shown in prototypes [33] to deliver frequency changes with overheads (lock times) of less than two cycles. The "high" and "low" frequencies are encoded locally on each core as multiplication factors. They are used to change the core frequency when directed by the *Booster* governor.

2.3. The Booster Governor

Cores are assigned dynamically to one of the two supply voltages according to a schedule controlled by the *Booster* governor. The governor is an on-chip programmable microcontroller similar to those used to manage power in the Intel Itanium [28] and Core i7 [15]. The governor can implement a range of boosting algorithms, depending on the goals for the system, such as mitigating frequency variation or reducing imbalance in parallel applications.

3. Booster VAR

The goal of *Booster VAR* is to maintain the same average per-core frequency across all cores in a CMP. To achieve this, the governor schedules cores that are inherently "slow" to spend more time on the higher V_{dd} line, improving their average frequency. Similarly, "fast" cores are assigned to spend more time on the low rail, saving power. The result is a heterogeneous CMP with homogeneous performance. The governor manages a "boost budget" that ensures chip power constraints (such as TDP) are not exceeded. For simplicity the "boost budget" is expressed in terms of maximum numbers of cores N_b that can be sped up at any given time. A boost schedule is chosen such that the average frequency for all the cores is the same over a predefined "boost interval."

3.1. VAR Boosting Algorithm

Booster VAR can be programmed to maintain a target CMP frequency from a range of possible frequencies. For instance, the target frequency can be set to the frequency achieved by the fastest core while on the low voltage rail. On each voltage rail, each core is set to run at its own best frequency, which is an integer multiple of the reference frequency F_r (e.g. multiples of 25MHz). Because of high variation, the maximum frequencies vary significantly from core to core. To keep track of each core's "execution progress" the *Booster* governor uses a set of counters. Each core's progress is represented by a value proportional to the number of cycles executed. Let MC_i represent one of the two clock multipliers (one for each voltage rail) selected for core i at the current time. Let PR_i represent the current progress metric of core i; in this case, number of cycles. To track progress of all cores, the governor will, at a frequency of F_r, increment PR_i by MC_i for each i. For instance, if the reference clock is 25MHz, and core 3 is currently running at a frequency of 300MHz, then every 40 nanoseconds, the governor will increment PR_3 by 12. (The counters are periodically reset to avoid overflow.)

The governor includes a *pace setter* counter that keeps track of the desired target frequency. The governor's job is to maintain the core progress counters as close as possible to the *pace setter*. At the end of each "boost interval," the governor selects the cores that have fallen behind the *pace setter* and boosts them during the next interval, with the restriction that no more than N_b cores can be boosted.

3.2. System Calibration

Booster VAR requires some chip-specific information that is collected post-manufacturing during the binning processes. The maximum frequencies of each core at the low and high voltages are determined through the regular binning process. This involves ramping up chip frequency by integer increments of the base frequency until all cores have exceeded their frequency limit. The high and low frequency multipliers for each core are recorded in ROM and are loaded into the governor during processor initialization.

4. Booster SYNC

The *Booster* framework can be used to compensate for other sources of performance variability such as work imbalance in shared-memory multithreaded applications. Parallel applications often have uneven workload distributions caused by algorithmic asymmetry, serial sections or unpredictable events such as cache misses [1, 3, 23]. This imbalance results in periods of little or no activity on some cores. To address application imbalance and improve execution efficiency, we developed *Booster SYNC*, which builds on the *Booster* framework.

4.1. Addressing Imbalance in Parallel Workloads

Booster SYNC reduces imbalance of multithreaded applications by favoring higher priority threads in the allocation of the "boost budget." *Booster SYNC*'s ability to very

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

quickly change the power state of each core allows it to react to changes in thread priority caused by synchronization events. *Booster SYNC* focuses on the four main synchronization primitives that are most common in commercial and scientific multithreaded workloads: *locks*, *barriers*, *condition waits*, and *starting and stopping threads*.

Barrier-based workloads divide up work among threads, execute parallel sections, and then meet again when that work is completed to synchronize and redistribute work. The primary inefficiencies of barrier-based workloads are imbalances in parallel sections, where some threads run longer than others, and sequential sections that cannot be parallelized. Speeding up threads that are still doing work while slowing down those blocked at the barrier should reduce workload imbalance, speed up the application and improve its efficiency.

Locks are used to acquire exclusive access to shared resources, and they are also often used to synchronize work and communicate across threads. Locks introduce two main inefficiencies. The first is caused by resource contention, which can stall execution on multiple threads. Another potential inefficiency occurs when locks are used for synchronization. For instance, locks are sometimes used to implement barrier-like functionality, with the same inefficiency issues as barrier. And locks are also often used to serialize thread execution. Reducing time spent by threads in the lock's "critical section" can potentially reduce both contention time and time spent in serialized execution.

Condition waits are a form of explicit inter-process communication, where a thread blocks until some other thread signals for it to continue executing. Among other things, conditions are often used in producer-consumer algorithms, where the consumer blocks until the producer signals that there is input available. To improve performance, blocked threads can give up their boost budget to speed up active cores.

Finally, some workloads dynamically spawn and terminate worker tasks. A core that is disabled because it has no task assigned is essentially the same as a core that is blocked, although it is possible to save slightly more power by turning power off completely. The boost budget of inactive cores can be redistributed to those cores that have work to do.

Unlike prior work that minimizes power for unbalanced workloads [1, 3, 23], our objective is to minimize runtime while remaining power-efficient. Also, unlike prior work we do not rely on criticality predictors to identify high-priority threads. Prediction would be too imprecise for lock and condition based workloads. Instead, *Booster SYNC* is a purely reactive system that uses hints provided by synchronization libraries and is managed by hardware to determine which cores are blocked and which ones are active.

Thread Progress	Thread Priority State
Thread spawned	*normal*
Thread terminated	*none (core off)*
Thread reaches barrier (not last)	*blocked*
Last thread reaches barrier	*normal* (all threads)
Lock acquire	*critical*
Lock release	*normal*
Block on lock	*blocked*
Block on condition	*blocked*
Condition signal	*normal*
Condition broadcast	*normal* (all threads)

Table 1. Thread priority states set by synchronization events.

4.2. Hardware-based Priority Management

Booster SYNC relies on hints from synchronization primitives to determine the states of all threads currently running. We define the following priority states for a thread: *blocked, normal*, and *critical*. When a thread is first spawned, it is set to *normal*. If a thread reaches a barrier, and is not the last one, its state is set to *blocked*. If it is the last thread to arrive at the barrier, it sets the state of all the other threads to *normal*. Conditions work in a similar way, so if a thread is blocked on a condition, its state is *blocked*. Threads that receive the condition signal/broadcast are set to *normal*. When a thread attempts to acquire a lock, there are two possible state transitions: if the thread acquires the lock, its state is set to *critical*, otherwise it is set to *blocked*. It is assumed that a critical section is likely to result in threads competing for a shared resource. Speeding up critical threads should reduce contention time, thus speeding up the whole application. Finally, when a thread terminates while there are no waiting threads in the run queue, a core will become idle and may be switched off. Thread priority states and transitions are summarized in Table 1.

The *Booster* governor keeps track of thread priorities. The priority state of each thread is stored as a 2-bit value in a *Thread Priority Table (TPT)* that is memory-mapped and accessible at process level. Priority tables are part of the virtual address space of each process, which allows any thread to change its own priority or the priorities of other threads belonging to the same process. Frequently updated TPT entries are cached in the local L1 data caches of each CPU for quick access.

The governor maintains TPT entries coherent with a *Core Priority Table (CPT)*, a centralized hardware table managed by the *Booster* governor and the OS. Note that multiple independent parallel processes can run on the CMP at the same time. The CPT is used as a cache for the TPT entries corresponding to the threads that are currently scheduled on the CMP, regardless of which process they belong to, as shown in Figure 3. Each CPT entry is tagged with the physical address of the corresponding TPT entry and acts as a direct-mapped cache with as many entries as there are

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

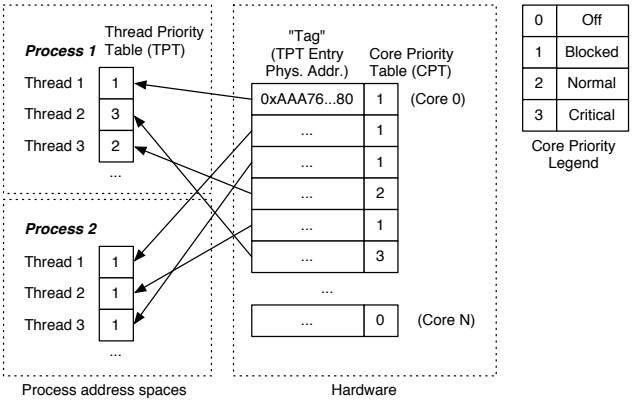

Figure 3. Thread Priority Tables are mapped into the process address space and cached in the Core Priority Table.

processors in the system. Each entry contains the priority value for the thread running on the corresponding core. The CPT entries are maintained coherent with local copies from each core through the standard cache coherence protocol.

4.3. SYNC Boosting Algorithm

Booster SYNC requires some minor changes to the boosting algorithm used in *Booster VAR* (Section 3.1). Just like in *Booster VAR*, the governor maintains a list of active cores sorted by core progress. In addition, *Booster SYNC* moves all *critical* threads to the head of the list. Given a "boost budget" of N_b cores *Booster SYNC* assigns the top N_b cores in the list to the high voltage rail. Cores that are in the *blocked* state are removed from the boost list and set to a low power mode (clock gated, on the low $V_{\rm dd}$). *Booster SYNC* will accelerate only *critical* and *normal* threads. If many threads are blocked, fewer than N_b may be boosted.

Booster SYNC uses the same core progress counters and metric as *Booster VAR*. However, progress of cores assigned *blocked* threads is accounted for differently. *Blocked* cores are removed from the boost list and their progress counters are no longer incremented by the governor. As a result, the progress counters of cores emerging from *blocked* states will indicate that they have fallen behind other cores. This would cause *Booster* to assign an excessive amount of boost to the previously-blocked threads. To avoid this issue, whenever a core changes state from *blocked* to *normal* or *critical*, its progress counter is set to the maximum counter value of all other active cores. This will place the core towards the bottom of the boost list.

4.4. Library and Operating System Support

Booster SYNC does not require special instrumentation of application software or special CPU instructions. Instead, it relies on modified versions of synchronization libraries that are typically supplied with the operating system, such as OpenMP and pthreads. To provide priority hints to the hardware, libraries write to entries in the TPT. When a running thread updates a local copy of a TPT entry, cache coherence will ensure that the CPT is also updated. Note that hints could be implemented in the kernel instead of the synchronization library, but the kernel is typically not informed as to which threads are holding locks (*critical*), limiting available TPT states to *normal* and *blocked*.

During initialization, a process makes system calls to inform the OS as to where its table entries are virtually located; the OS translates these into physical addresses and tracks this as part of the process and thread contexts. Association of TPT and CPT entries is also handled by the OS. On a context switch, the OS updates the CPT tag for each core with the physical address of the TPT entry of the corresponding thread. The OS also guarantees protection and isolation for CPT entries belonging to different processes.

4.5. Other Workload Rebalancing Solutions

In our implementation, *Booster* uses cycle count as a metric of core progress. This allows *Booster VAR* to ensure that all cores execute the same number of cycles over a finite time interval. However, by altering the way we track core progress, we can use the *Booster* framework to support other solutions for addressing workload imbalance. For instance, Bhattacharjee and Martonosi [1] observed that for instruction-count-balanced workloads, imbalance is caused by divergent L2 miss rates. *Booster* could reduce this imbalance by using retired instruction count as the execution progress metric. This will, in effect, speed up threads that suffer more long latency cache misses and help them keep up with the rest of the threads. Another alternative progress metric might be explicit markers inserted by the programmer or compiler into the application, as in [3]. We leave detailed exploration of these approaches to future work.

5. Evaluation Methodology

5.1. Architectural Simulation Setup

We used the SESC [32] simulator to model a 32-core CMP. Each core is a dual-issue out-of-order architecture. The *Linux Threads* library was ported to SESC in order to run the PARSEC benchmarks that require the pthreads library. We ran the PARSEC benchmarks (*blackscholes, bodytrack, fluidanimate, swaptions, dedup,* and *streamcluster*) and SPLASH2 benchmarks (*barnes, cholesky, fft, lu, ocean, radiosity, radix, raytrace,* and *water-nsquared*) with the sim-small and reference input sets.

We collected runtime and activity information, which we use to determine energy. Energy numbers are scaled for supply voltage, technology and variation parameters.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

5.2. Delay, Power and Variation Models

For power and delay models at near threshold, we use the models from Markovìc et al [26], reproduced here in Equations 1, 2, 3, 4 and 5.

$I_{\rm ds}$ is the drain-source current used to compute dynamic power. $I_{\rm Leakage}$ is the leakage current used to compute static power. IC is a parameter called the inversion coefficient that describes proximity to threshold voltage, η is the sub-threshold slope, μ is the carrier mobility, and $k_{\rm fit}$ and $k_{\rm tp}$ are fitting parameters for current and delay.

$$I_{\rm ds} = \frac{I_{\rm s} \cdot {\rm IC}}{k_{\rm fit}} \quad (1)$$

$$I_{\rm s} = 2 \cdot \mu \cdot C_{\rm ox} \cdot \frac{W}{L} \cdot \phi_t^2 \cdot \eta \quad (2)$$

$${\rm IC} = \left(\ln \left({\rm e}^{\frac{(1+\sigma)\cdot V_{\rm dd} - V_{\rm th}}{2 \cdot \eta \cdot \phi_t}} + 1 \right) \right)^2 \quad (3)$$

$$t_p = \frac{k_{\rm tp} \cdot C_{\rm Load} \cdot V_{\rm dd}}{2 \cdot \eta \cdot \mu \cdot C_{\rm ox} \cdot \frac{W}{L} \cdot \phi_t^2} \cdot \frac{k_{\rm fit}}{\rm IC} \quad (4)$$

$$I_{\rm Leakage} = I_{\rm s} \cdot {\rm e}^{\frac{\sigma \cdot V_{\rm dd} - V_{\rm th}}{\eta \cdot \phi_t}} \quad (5)$$

We model variation in threshold voltage ($V_{\rm th}$) and effective gate length ($L_{\rm eff}$) using the VARIUS model [34]. We used the Markovìc models to determine core frequencies as a function of $V_{\rm dd}$ and $V_{\rm th}$. To model the effects of $V_{\rm th}$ variation on core frequency, we generate a batch of 100 chips that have different $V_{\rm th}$ (and $L_{\rm eff}$) distributions generated with the same mean and standard deviation. This data is used to generate probability distributions of core frequency at nominal and near threshold voltages.

To keep simulation time reasonable, we ran the microarchitectural simulations using four random normal distributions of core $V_{\rm th}$ with a standard deviation of 12% over the nominal $V_{\rm th}$. All core and cache frequencies are integer multiples of a 25MHz reference clock. The L2 cache and NoC are on the lower voltage rail, with operating frequencies constrained accordingly. We ran all experiments with each frequency distribution, and we report the arithmetic mean of the results.

Table 2 summarizes the experimental parameters.

CMP architecture	
Cores	32, out-of-order
Fetch/issue/commit width	2/2/2
Register file size	76 int, 56 fp
Instruction window	56 int, 24 fp
L1 data cache	4-way 16KB, 1-cycle access
L1 instruction cache	2-way 16KB, 1-cycle access
Distributed L2 cache	8-way 8MB, 10 cycle access
Technology	32nm
Core, L1 $V_{\rm dd}$	400-600mV
Core, L1 frequency	300-2300MHz, 25MHz increments
L2, NoC $V_{\rm dd}$	400mV
L2, NoC frequency	400MHz
Variation parameters	
$V_{\rm th}$ mean (μ),	210mV
$V_{\rm th}$ std. dev./mean (σ/μ)	12%

Table 2. Summary of the experimental parameters.

$V_{\rm th}$ σ/μ	Freq. σ/μ at 900mV	Freq. σ/μ at 400mV
3%	1.0%	7.5%
6%	2.1%	15.1%
9%	3.2%	22.8%
12%	4.4%	30.6%

Table 3. Frequency variation as a function of $V_{\rm th}$ σ/μ and $V_{\rm dd}$.

6. Evaluation

We evaluate the performance and energy benefits of eliminating core-to-core frequency variation with *Booster VAR* and reducing application imbalance with *Booster SYNC*. We compare the effectiveness of *Booster VAR* to a mechanism that mitigates frequency variation through thread scheduling similar to the ones in [31, 36]. We also compare *Booster SYNC* with an ideal implementation of Thrifty Barrier [23].

We begin by evaluating the effects of process variation on core frequency at low voltage.

6.1. Frequency Variation at Low Voltage

Low-voltage operation increases the effects of process variation dramatically. Using our variation model, we examine within-die frequency variation at both nominal (900mV) and near threshold $V_{\rm dd}$ (400mV). In Figure 4 we show core-to-core variation in frequency as a probability distribution of core frequency divided by die mean (average over all cores in the same die). The distributions shown are for 9% and 12% within-die $V_{\rm th}$ variation. At nominal $V_{\rm dd}$ the distribution is relatively tight, with only 4.4% frequency standard deviation divided by the mean (σ/μ). At low voltage, frequency variation is 30.6% σ/μ with cores deviating from less than half to more than $1.5\times$ the mean. Table 3 summarizes the impact of different amounts of $V_{\rm th}$ variation on frequency σ/μ.

The high within-core variation deteriorates CMP frequency significantly. In the absence of variation, a 32nm CMP at 400mV would be expected to run at about 400MHz. At the same $V_{\rm dd}$, a 12% $V_{\rm th}$ variation would bring the average frequency across all dies to 149MHz, assuming each die's frequency is set to that of its slowest core.

To avoid the severe degradation in CMP frequency, each core can be allowed to run at its best frequency, resulting in a heterogeneous CMP. However, the random nature of variation-induced heterogeneity can still lead to poor and unpredictable performance.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

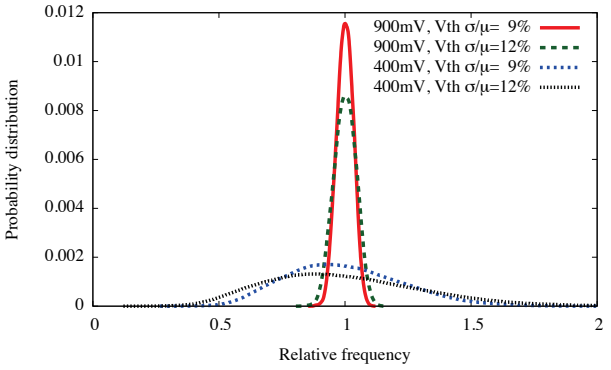

Figure 4. Core-to-core frequency variation at nominal and near-threshold V_{dd}, relative to die mean (average over all cores in the same die).

6.2. Workload Balance in Parallel Applications

The way in which parallel applications handle workload partitioning has a direct impact on their performance when running on heterogeneous vs. homogeneous CMPs. Broadly speaking, parallel applications divide work either statically at compile time or dynamically during execution.

6.2.1 Static Load Partitioning

Statically partitioned workloads are generally designed for homogeneous systems. Significant effort goes into making sure work assignment is as balanced as possible. In general, well-balanced workloads are expected to perform poorly on heterogeneous CMPs because their performance is limited by the slowest core. For instance, each thread of *fft* executes the same algorithm and processes the same amount of data. A slow thread bottlenecks the performance of the entire application. These applications should benefit from the performance homogeneity of *Booster VAR*.

Many applications like *lu*, *radix*, and *dedup* are inherently unbalanced due to algorithmic characteristics. In theory, these applications could perform well on heterogeneous systems if critical threads are continuously matched to fast cores. In practice, their performance is unpredictable, especially when running on systems with variation-induced heterogeneity. These are the types of applications we expect will benefit most from *Booster SYNC*.

6.2.2 Dynamic Load Balancing

Some applications, like *radiosity* and *raytrace*, employ mechanisms for dynamically rebalancing workload allocation across threads. Dynamic load balancing is beneficial when the runtime of individual work units is highly variable. These applications adapt well to performance-heterogenous systems. As a result, we expect them to benefit little from the *Booster* framework.

We summarize in Table 4 the relevant algorithmic characteristics of all benchmarks we simulated. We include the expected benefits from *Booster VAR* and *Booster SYNC*. For applications like *radix*, *water-nsquared*, *fluidanimate* and *bodytrack*, even though they are either statically partitioned and balanced, or use dynamic load balancing, some benefit from *Booster SYNC* is still possible. This is because the applications include some amount of serialization in the code or have a serial master thread that can be sped up by *Booster SYNC*.

6.3. Booster Performance Improvement

We evaluate the performance of *Booster VAR* and *Booster SYNC* relative to a heterogenous baseline in which each core runs at its best frequency. Figure 5 shows the execution times of all benchmarks normalized to the baseline ("Heterogeneous"). The target frequency for *Booster* is chosen to match the average frequency of the heterogeneous baseline.

We also compare *Booster VAR* and *Booster SYNC* to a heterogeneity-aware thread scheduling approach, "Hetero Scheduling," that dynamically migrates slow threads to faster cores and short-running threads to slower cores. This technique is similar to those used to cope with heterogeneity in [31] and [36], but we apply it to multithreaded workloads. In our implementation, migration occurs at barrier synchronization points using thread criticality information collected over the previous synchronization interval. We chose an ideal implementation of "Hetero Scheduling" that introduces no performance penalty from thread migration, except when caused by incorrect criticality prediction from one barrier to the next.

Booster VAR improves the performance of workloads that use static work allocation by an average of 14% compared to the baseline. "Hetero Scheduling" also performs better than the baseline for statically scheduled workloads but reduces execution time by only 5%. As expected, workloads that use dynamic rebalancing adapt well to heterogeneity and have no performance benefit from *Booster VAR* or from "Hetero Scheduling." *Booster VAR* is especially beneficial for balanced workloads such as *fft*, *blackscholes* or *water-nsquared* that are hurt by heterogeneity. "Hetero Scheduling," on the other hand, can do little to help these cases.

Booster SYNC builds on the *Booster VAR* framework, allocating the boost budget to critical or active threads. This leads to significant performance improvements, even for workloads where *Booster VAR* is ineffective. For statically partitioned workloads with significant imbalance, such as *dedup*, *swaptions* or *streamcluster*, *Booster SYNC* improves performance between 15% and 20%. *Booster VAR* brings no performance gains for these applications. *Booster SYNC* also helps some dynamically balanced applications that

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

Benchmark	*Workload characteristics*	*Booster VAR*	*Booster SYNC*
barnes	Static partitioning of data, balanced	Likely to benefit	Unlikely to benefit
cholesky	Static partitioning of data, no global synchronization	Likely to benefit	Unlikely to benefit
fft	Static partitioning of data, highly balanced	Likely to benefit	Unlikely to benefit
lu	Static partitioning of data, highly unbalanced	Unpredictable	Likely to benefit
ocean	Static partitioning of data, balanced, heavily synchronized	Likely to benefit	Unlikely to benefit
radiosity	Task stealing and dynamic load balancing	Unlikely to benefit	Unlikely to benefit
radix	Static partitioning of data, balanced, some serialization	Likely to benefit	Possible benefit
raytrace	Task stealing and dynamic load balancing	Unlikely to benefit	Unlikely to benefit
volrend	Task stealing and dynamic load balancing	Unlikely to benefit	Unlikely to benefit
water-nsquared	Static partitioning of data, balanced, some serialization	Likely to benefit	Possible benefit
blackscholes	Static partitioning of work, balanced	Likely to benefit	Unlikely to benefit
bodytrack	Serial master, dynamically balanced parallel kernels	Unlikely to benefit	Possible benefit
dedup	Unbalanced software pipeline stages with multiple thread pools	Unpredictable	Likely to benefit
fluidanimate	Static partitioning of work, balanced, some serialization	Likely to benefit	Possible benefit
streamcluster	Static partitioning of data, unbalanced, heavily synchronized	Unpredictable	Likely to benefit
swaptions	Static partitioning of data, unbalanced	Unpredictable	Likely to benefit

Table 4. Benchmark characteristics and expected benefit from *Booster* given algorithm characteristics

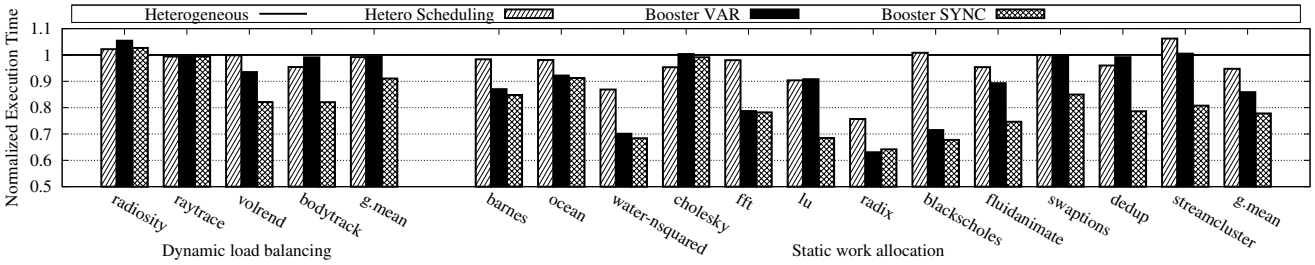

Figure 5. Runtimes of *Booster VAR*, *Booster SYNC*, and "Hetero Scheduling," relative to Heterogeneous (best frequency) baseline.

have significant serialization due to resource contention, such as *bodytrack*, by boosting their critical sections.

Balanced applications like *fft*, *blackscholes* and *water-nsquared*, which benefit significantly from *Booster VAR*, have little or no additional performance gains from *Booster SYNC*. Overall, *Booster SYNC* complements *Booster VAR* very well. On average, it is 22% faster than the baseline for static workloads and 9% faster for dynamic workloads.

6.3.1 Impact of Different Synchronization Primitives

Figure 6 shows the effects of *Booster SYNC* responding to hints from different synchronization primitives in isolation, for a few benchmarks. *lu* is a very unbalanced barrier-based workload. Providing the *Booster* governor with hints about barrier activity speeds up the application by 24% over *Booster VAR*. Information about locks, conditions or thread spawning does not help speed up *lu*. *bodytrack* makes heavy use of locks, with a substantial amount of contention. Speeding up critical sections results in a 17% speed increase over *Booster VAR*. Boosting cores that are not blocked on condition waits also helps. *swaptions* uses no synchronization at all but instead actively spawns and terminates worker threads. As a result, it benefits greatly from providing the *Booster* governor with information about active thread count, which allows the redistributing of boost budget from unused cores. This speeds up *swaptions* by 15% over *Booster VAR*.

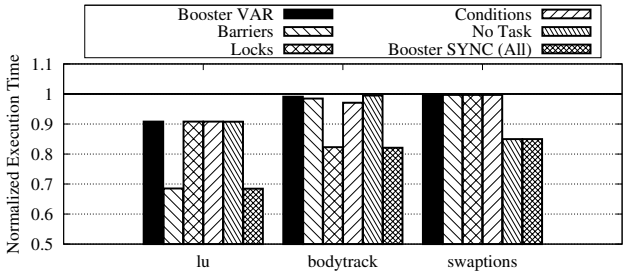

Figure 6. *Booster SYNC* performance impact of using hints from different types of synchronization primitives in isolation.

6.4. Booster Energy Delay Reduction

We examine the energy implications of *Booster VAR* and *Booster SYNC* compared to the baseline. Figure 7 shows the energy delay product (ED) for each benchmark. We compare with an ideal implementation of "Thrifty Barrier" [23], which puts cores into a low-power state when they reach a barrier, with no wakeup time penalty.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

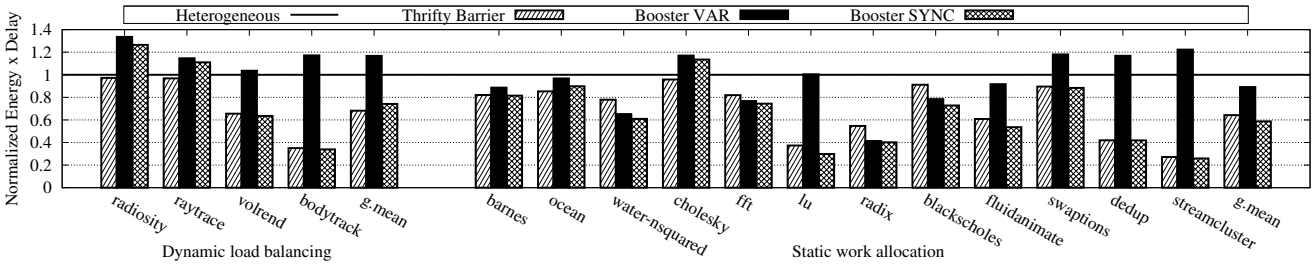

Figure 7. Energy×delay for *Booster VAR*, *Booster SYNC*, and ideal Thrifty Barrier, relative to Heterogeneous (best frequency) baseline.

Booster VAR generally uses more power than the "Heterogeneous" baseline in order to achieve homogeneous performance at the same average frequency. As a result, ED is actually higher than the baseline for the dynamically balanced workloads. However, for statically partitioned benchmarks, *Booster VAR* lowers ED by 11%, on average. *Booster SYNC* is much more effective at reducing energy delay because in addition to speeding up applications, it saves power by putting inactive cores to sleep. It achieves 41% lower ED for static workloads and 25% lower ED for dynamic workloads, relative to the baseline.

Our implementation of "Thrifty Barrier" has considerably lower ED than *Booster VAR* because it runs on a lower-power baseline and, unlike *Booster VAR*, it has the ability to put inactive cores into a low power mode. The ED of *Booster SYNC* is close to that of the ideal "Thrifty Barrier" implementation: slightly higher for dynamic workloads and slightly lower for static workloads. Note that the goals for *Booster* and "Thrifty Barrier" are different. *Booster* is meant to improve performance while "Thrifty Barrier" is designed to save power.

6.5. Booster Performance Summary

Figure 8 summarizes the results, showing geometric means across all benchmarks. All results are normalized to the "Heterogeneous" (best frequency) baseline. In addition, we also compare to a more conservative design, "Homogeneous," in which the entire CMP runs at the frequency of its slowest core. To make a fair comparison, we assume the voltage of the "Homogeneous" CMP is higher, such that its frequency is equal to the average frequency of the "Heterogeneous" design.

The frequency for the "Homogeneous" baseline is the same as the target frequency for *Booster VAR*. As a result, the execution time of the two is very close, with *Booster VAR* only slightly slower due to the overhead of the *Booster* framework. However, to achieve the same frequency, the "Homogeneous" baseline runs at a much higher voltage, which increases power consumption by 70% over the "Heterogeneous" baseline. *Booster VAR* also has higher power than the heterogeneous baseline, but by only 20%. *Booster SYNC* is a net gain in both performance (19% faster than

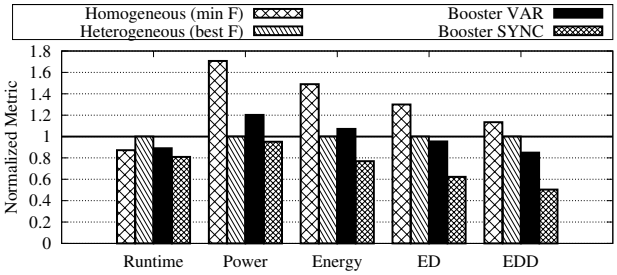

Figure 8. Summary of performance, power and energy metrics for *Booster VAR* and *Booster SYNC* compared to the "Homogeneous" and "Heterogeneous" baselines.

baseline) and power (5% lower than baseline), which leads to 23% lower energy and 38% lower energy delay product.

When considering the "voltage-invariant" metric ED^2, *Booster VAR* is 16% better and *Booster SYNC* is 50% better than the heterogeneous baseline.

7. Related Work

7.1. Low Voltage Designs

Previous work has demonstrated the energy efficiency of very low voltage designs [6, 8, 9, 26, 39]. Architectures designed specifically to take advantage of low voltage properties such as fast caches relative to logic have been proposed by Zhai et al. [39] and Dreslinski et al. [9]. Other work has focused on improving the reliability of large caches in low voltage processors [11, 29]. While significant progress has been made in bringing this technology to market, including a prototype processor from Intel [38], many challenges remain, including reliability and high variation.

7.2. Dual-Vdd Architectures

Previous work has proposed dual and multi-V_{dd} designs with the goal of improving energy efficiency. Most previous work has focused on tuning the delay vs. power-consumption of paths at fine granularity within the processor. For instance, Kulkarni et al. [20] propose a solution for assigning circuit blocks along critical paths to the higher power supply, while blocks along non-critical paths are as-

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

signed to a lower power supply. Liang et al. proposed Revival [24], which uses voltage selection at pipeline stage granularity to reduce the effects of delay variation. Calhoun and Chandrakasan proposed local voltage dithering [4] to achieve very fast dynamic voltage scaling in subthreshold chips. These solutions assign multiple voltages at much finer granularity than in our design, incurring a higher design and verification complexity.

Miller et al. [30] proposed using dual-V_{dd} assignment at core granularity to reduce variation effects. Based on manufacturing time test results, fast cores are placed on a low voltage rail (to reduced wasted power) and slow cores on a higher rail (to speed them up). This static assignment reduces frequency variation but does not eliminate it completely. The *Booster* framework uses dynamic voltage assignment, which is much more effective, eliminating frequency variation completely.

In his dissertation [7], Dreslinski proposed a dual V_{dd} system for fast performance boosting of serial bottlenecks in NTC systems. This was specifically applied to overcoming challenges with parallelizing transactional memory systems and to throughput computing. Dreslinski's work boosts cores to very high frequency, at nominal voltages, with much higher power cost. In *Booster*, both V_{dd} rails are at low voltage, improving the system's energy efficiency. *Booster* also eliminates frequency variation.

7.3. On-chip Voltage Regulators

Fast on-chip regulators [18, 19] are a promising technology that could allow fine-grain voltage and frequency control at core (or clusters of cores) granularity. They can also perform voltage changes much faster than off-chip regulators, making them a more flexible alternative to a dual-V_{dd} design. However, on-chip regulators do face significant challenges to widespread adoption. One challenge is low efficiency, with power losses of 25–50% due to their high switching frequency. They are also more susceptible to large voltage droops because of much smaller decoupling and filter capacitances available on-chip. Limiting the size of on-chip capacitors and inductors without affecting voltage stability remains challenging, although significant progress has been made in recent work [18].

7.4. Balancing Parallel Applications

Previous work has exploited imbalance in multithreaded parallel workloads primarily by scaling the supply voltage and frequency of processors running non-critical threads. Thrifty Barrier [23] uses prediction of thread runtime to estimate how long a thread will wait at a barrier. For longer sleep times, the CPU can be put into deeper sleep states that may require more time to wake up. An alternative to sleeping at the barrier is proposed by Liu et al. [25]. Their approach is to use DVFS to slow down non-critical threads so that all threads complete at the same time. This approach has the potential for greater energy savings because non-critical threads run at a lower average voltage and frequency, which, in general, is more energy-efficient then running at a high voltage and frequency and then going into sleep mode. Cai et al. take a different approach to criticality prediction in Meeting Points [3]. They use explicit instrumentation of worker threads to keep track of progress and use this information to decide on voltage and frequency assignments.

Our work is different from these previous designs in two important ways. First, our goal is to improve performance whereas in the work described above the goal was to save power. Second, our approach is reactive adaptation, which means we do not require predictors of thread criticality. While we do use hints from the synchronization libraries to determine thread priority, because *Booster SYNC* is entirely reactive, these hints can be simple notifications about state changes rather than complex and sometimes inaccurate predictions.

Task stealing [2] is a popular scheduling technique for fine-grain parallel programming models. Task stealing poses several challenges in terms of organizing the task queues (distributed or hierarchical), choosing a policy for enqueuing, dequeuing or stealing tasks, etc. It has also been shown [10, 12] that no single task stealing solution works for all scheduling-sensitive workloads. The *Booster* framework is less helpful to parallel applications that use dynamic work allocation such as task stealing.

8. Conclusions

This paper presents *Booster*, a simple, low-overhead framework for dynamically reducing performance heterogeneity caused by process variation and application imbalance. *Booster VAR* completely eliminates core-to-core frequency variation resulting in improved performance for statically partitioned workloads. *Booster SYNC* reduces the effects of workload imbalance, improving performance by 19% on average and reducing energy delay by 23%.

Acknowledgements

This work was supported in part by the National Science Foundation under grant CCF-1117799 and an allocation of computing time from the Ohio Supercomputer Center. The authors would like to thank the anonymous reviewers for their valuable feedback and suggestions, most of which have been included in this final version.

References

[1] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 290–301, June 2009.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, September 1999.

[3] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 240–249, October 2008.

[4] B. Calhoun and A. Chandrakasan. Ultra-dynamic voltage scaling (UDVS) using sub-threshold operation and local voltage dithering. 41(1):238–245, January 2006.

[5] A. Chandrakasan, D. Daly, D. Finchelstein, J. Kwong, Y. Ramadass, M. Sinangil, V. Sze, and N. Verma. Technologies for ultradynamic voltage scaling. *Proceedings of the IEEE*, 98(2):191–214, February 2010.

[6] L. Chang, D. Frank, R. Montoye, S. Koester, B. Ji, P. Coteus, R. Dennard, and W. Haensch. Practical strategies for power-efficient computing technologies. *Proceedings of the IEEE*, 98(2):215–236, February 2010.

[7] R. Dreslinski. *Near Threshold Computing: From Single Core to Many-Core Energy Efficient Architectures*. PhD thesis, The University of Michigan, 2011.

[8] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, February 2010.

[9] R. G. Dreslinski, G. K. Chen, T. Mudge, D. Blaauw, D. Sylvester, and K. Flautner. Reconfigurable energy efficient near threshold cache architectures. In *International Symposium on Microarchitecture*, pages 459–470, December 2008.

[10] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *International Conference on OpenMP in a New Era of Parallelism*, pages 100–110, May 2008.

[11] H. R. Ghasemi, S. Draper, and N. S. Kim. Low-voltage on-chip cache architecture using heterogeneous cell sizes for multi-core processors. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 38–49, February 2011.

[12] Y. Guo, R. Barik, R. Raman, and V. Sarka. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, May 2009.

[13] S. Herbert and D. Marculescu. Mitigating the impact of variability on chip-multiprocessor power and performance. *IEEE Transactions on Very Large Scale Integrated Systems*, 17:1520–1533, October 2009.

[14] E. Humenay, D. Tarjan, and K. Skadron. The impact of systematic process variations on symmetrical performance in chip multiprocessors. In *Design, Automation and Test in Europe*, April 2007.

[15] Intel Core™ i7 Processor. http://www.intel.com.

[16] International Technology Roadmap for Semiconductors (2009).

[17] H. Jiang and M. Marek-Sadowska. Power gating scheduling for power/ground noise reduction. In *Design Automation Conference*, pages 980–985, June 2008.

[18] W. Kim, D. Brooks, and G.-Y. Wei. A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation. In *International Solid-State Circuits Conference*, pages 268–270, February 2011.

[19] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 123–134, February 2008.

[20] S. Kulkarni, A. Srivastava, and D. Sylvester. A new algorithm for improved VDD assignment in low power dual VDD systems. In *International Symposium on Low Power Electronics and Design*, pages 200–205, May 2004.

[21] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, and R. Kumar. Next generation Intel Core micro-architecture (Nehalem) clocking. *IEEE Journal of Solid-State Circuits*, 44(4):1121–1129, April 2009.

[22] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, 8:48–51, July 2009.

[23] J. Li, J. Martínez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 14–24, February 2004.

[24] X. Liang, G.-Y. Wei, and D. Brooks. ReVIVaL: A variation-tolerant architecture using voltage interpolation and variable latency. *IEEE Micro*, 29(1):127–138, 2009.

[25] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–5, April 2005.

[26] D. Markovic, C. Wang, L. Alarcon, T.-T. Liu, and J. Rabaey. Ultralow-power design in near-threshold region. *Proceedings of the IEEE*, 98(2):237–252, February 2010.

[27] P. Maulik and D. Mercer. A DLL-based programmable clock multiplier in 0.18-um CMOS with -70 dBc reference spur. *IEEE Journal of Solid-State Circuits*, 42(8):1642–1648, August 2007.

[28] R. McGowen, C. A. Poirier, C. Bostak, J. Ignowski, M. Millican, W. H. Parks, and S. Naffziger. Power and temperature control on a 90-nm Itanium family processor. *IEEE Journal of Solid-State Circuits*, 41(1):229–237, January 2006.

[29] T. Miller, J. Dinan, R. Thomas, B. Adcock, and R. Teodorescu. Parichute: Generalized turbocode-based error correction for near-threshold caches. In *International Symposium on Microarchitecture*, pages 351–362, December 2010.

[30] T. Miller, R. Thomas, and R. Teodorescu. Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed stages. *IEEE Computer Architecture Letters*, 11(1), 2012.

[31] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: Fine-grained power management for multi-core systems. In *International Symposium on Computer Architecture*, pages 302–313, June 2009.

[32] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[33] T. Saeki, M. Mitsuishi, H. Iwaki, and M. Tagishi. A 1.3-cycle lock time, non-PLL/DLL clock multiplier based on direct clock cycle interpolation for clock on demand. *IEEE Journal of Solid-State Circuits*, 35(11):1581–1590, November 2000.

[34] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. VARIUS: A model of parameter variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13, February 2008.

[35] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Operating Systems Review*, 43(2):66–75, April 2009.

[36] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *International Symposium on Computer Architecture*, pages 363–374, June 2008.

[37] J. Torrellas. Architectures for extreme-scale computing. *IEEE Computer*, 42:28–35, November 2009.

[38] S. Vangal. A solar powered IA core? No way! Research@Intel, September 2011. http://blogs.intel.com/research/2011/09/ntvp.php.

[39] B. Zhai, R. G. Dreslinski, D. Blaauw, T. Mudge, and D. Sylvester. Energy efficient near-threshold chip multi-processing. In *International Symposium on Low Power Electronics and Design*, pages 32–37, August 2007.

Authorized licensed use limited to: The Ohio State University. Downloaded on November 13,2024 at 01:56:59 UTC from IEEE Xplore. Restrictions apply.