UTILIZING CRITICALITY STACKS FOR DYNAMIC VOLTAGE AND FREQUENCY SCALING

A Thesis

by

LEE BRYAN ELLIOTT

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Paul Gratz
Committee Members,	Weiping Shi
	Ricardo Bettati
Head of Department,	Miroslav M. Begovic

August 2018

Major Subject: Computer Engineering

Copyright 2018 Lee Bryan Elliott

ABSTRACT

Thread imbalance is inevitable for multithreaded applications due to the necessity of synchronization primitives to coordinate access to memory and system resources. This imbalance leads to a bounding of application performance, but, more importantly for mobile devices, this imbalance also leads to energy inefficiencies. Recent works have begun to quantify this imbalance and look to leverage it not only for performance improvements, but for energy savings as well. All these works, though, test the theory through the use of simulators and power estimation tools. These results may show that the theory is sound, but the complexities of how a real machine handles synchronization may lead to diminished results by either having too large of a performance impact, or too little energy savings. In this work, we implement one such algorithm, PCSLB, and improve upon it in order to see if the results shown for this technique are feasible for use in real machines. With the improved algorithm, PCSLB-Max, and the CritScale Linux kernel module, we show that, in fact, there are energy saving available to us while mitigating the performance.

DEDICATION

To my family and friends.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Dr. Paul Gratz for supporting me throughout my research and providing valuable input. I would also like to thank Dr. Weiping Shi and Dr. Riccardo Bettati for serving on my committee and providing useful feedback on my work.

I would take this opportunity to acknowledge all the help I have received from my friends and faculty during my time at Texas A&M University. I will always cherish the memorable times I had with my friends working on various course projects and extensive learning I gained from discussions with my faculty.

NOMENCLATURE

CMP	Chip Multi-Processor
MPSOC	Multi-Processor System-On-Chip
DVFS	Dynamic Voltage/Frequency Scaling
TLP	Thread-Level Parrallelism
PCSLB	Predicting thread Criticality and frequency Scalability for active Load Balancing
HWP	Hardware-coordinated P-States
V	Voltage
F	Frequency

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Performance growth has been the focus for the industry for many years, mainly driven by transistor density improvements from generation to generation. Because of Dennard Scaling[7], performance per watt improved alongside these process changes. As a result, for much of the 1990's, a processor frequency was the sole metric for performance. This was the beginning of the Megahertz wars, where processors were evaluated solely on the frequency to determine the best computer systems. A worrying side-effect of this trend was that power dissipation was increased as well. By the early 2000's this trend became unsustainable, causing the paradigm shift to the Megawatts wars where power became the primary constraint for designs. With caps set on frequency due to power constraints, designers shifted to inclusion of multiple cores in a single package and eventually on a single die. This began the modern Chip Multi-Processor (CMP) technology.

In these designs, rather than a single high performance, and high power draw, core, many lower performance and power cores are paired together. Developers started using thread level parallelism to exploit the new performance of CMP designs, but that required threads to be able to communicate when then needed exclusivity over memory, so synchronization frameworks were used to synchronize threads and their access to shared resources. The downside to using synchronization constructs, such as mutual exclusion constructs (mutex), is that when one thread is in a critical section of code, any other thread that needs to access the shared resources being used must wait for the current thread to finish and unlock the mutex before it can proceed. Normally this means a thread that is blocked by a mutex must perform a no-op spin until the resource becomes available, wasting energy in the process. Modern schedulers in operating systems can be leveraged to schedule in other threads, or other processes while the blocked thread sleeps to ensure the machine is always performing useful work. These critical sections create syntonization issues due to not all threads being able to end at the same time, instead, barriers are used in the parent thread to prevent progression until all child threads have finished their work. The result of using barriers is that the program's runtime is now bounded by the time it takes to complete the lowest thread. These bottleneck threads, as stated by Amdhal's Law[28], determine the speedup that can be achieved, in addition to being the largest contributor to wasted energy as the other threads spin at the barrier.

1.1 Predicting Thread Criticality

Load imbalances due to synchronization constructs can result in both loss of performance and energy waste. The threads that cause this imbalance are termed critical threads. The goal of predicting thread criticality is an attempt to identify the critical thread in a workflow and opening up various optimization techniques for both power and performance. This critical thread can be run on a core of a higher frequency[2], migrated to a high-performance core[33] in the case of a heterogeneous architecture and providing it a prioritized resource allocation in a shared resource environment[10]. The focus for this research is on implementation of a per-core DVFS scheme to determine voltage/frequency operating points, as is a while held technique[16][32]. Identified critical threads can be run a higher frequencies for performance increases, and non-critical threads can be run at a lower frequency to conserve energy. Depending on program behavior, multiple different threads can become critical over the course of a program,s lifetime. Given this dynamic nature of identifying a critical thread, a dynamic algorithm is required to correctly identify the critical thread in a given time frame. DuBois, et. al., in [9], proposed a dynamic algorithms that looks at the time spent on useful work by a thread and the number of threads that may be blocked from completing work due to that thread to identify if that thread is critical or not. This method, while successful in identifying a critical thread, does not account for how a thread,s performance is impacted by various voltage/frequency points.

1.2 Predicting Thread Scalability

Not all threads behave similarly to changes in frequency. A threads performance in relation to the frequency of the core it is running on is referred to as that threads scalability. As identified by David, et. al.[20], and other works[27],[6], the execution of a thread can be divided into scalable ($T_{scalable}$) and non-scalable ($T_{non\_scalable}$) phases. The over all scalability of the thread can be defined as follows:

$$Scalability(S) = \frac{T_{\text{scalable}}}{T_{\text{scalable}} + T_{\text{non\_scalable}}} \quad 0 < \text{S} < 1 \tag{1.1}$$

Scalable phases are denoted by the workload being CPU-bound, resulting in the change in performance that is close to linear to the change in core frequency, while non-scalable phases are denoted by a workload that is memory intensive behavior due to off-core accesses resulting in changes in performance being independent of frequency changes. Due to this variability in scalability over time, an accurate dynamic algorithm for predicting the scalability of a thread is needed to ensure we aren't wasting energy by increasing the frequency of a thread that cannot take advantage of it.

Girdhar, et. al.[13], investigated this problem and was able to empirically derive a formula tracking L1 Data and L2 cache misses as inputs for the following formula:

$$S = \alpha_1 + \alpha_2 * L1D_{MPKI} + \alpha_3 * L2_{MPKI} \tag{1.2}$$

1.3 Motivation for Real Machine Testing

Up to this point, the focus of tests using Criticality-based DVFS has focused on simulations and power estimation tools. With simulators, it is possible to tweak parameters that are not possible with a real machine. Few results for DVFS schemes, outside of posts and discussion on Linux message boards, have been published using a real machine. Simulators, unlike real machines, execute code in a vacuum, letting the code run without any preemption from the scheduler, interrupts, or based on the needs of other processes. While this may give ideal results to test the theory, they may not represent what would happen in a real-world environment of modern computer systems. Since version 2.5.7, the Linux kernel introduced "Fast User-space Mutex",*futexes*, which will sleep a processes if it is unable to acquire the lock. This allows for the scheduler to schedule in a new process that is needing to be executed while the locking processes waits a wakeup signal to reattempt to acquire the lock[12]. If no process is available, the core will enter a sleep state to reduce power consumption. This differs from a simulator where a blocked thread can be seen as only spinning, thus wasting energy while being blocked. This means what energy is ŞwastedŤ in a simulator is actually not wasted in a real machine. Because of this important difference, the actual energy savings of a DVFS method may not appear in the real world.

In this work we will investigate the feasibility of implementation of Girdhar's PCSLB algorithm as a DVFS module in the Linux operating system to look at how a real system would behave given the overhead of the calculations need to perform the algorithms laid out by prior works. As such the contributions of this work are as follows:

- We develop a module to implement Girdhar's PCSLB algorithm for testing on a real machine.
- We investigate causes of any impact to performance
- We investigate methods for mitigating the impact of performance loss
- We present the performance, power and energy results and compare them with common Linux frequency governors used today.

The rest of the thesis are outlined as follows. Chapter 2 will give a literature survey of prior work on DVFS, thread criticality prediction, and scalability prediction. Chapter 3 will discuss the methods of implementation of the PCSLB algorithm using a custom Linux kernel module. Chapter 4 will go over improvements to the PCSLB algorithm. In Chapter 5, we present the experimental setup and results showing the performance and energy analysis of the CritScale module in comparison to prior work [13] and common Linux frequency governors in use today. At the end, in Chapter 6, will present the Conclusion and future work possible.

2. PRIOR WORK

Optimizing power and performance has been an important research topic for many years. Researchers have attempted to use a variety of Dynamic Voltage Frequency Scaling methods to gain performance while aiming to save power on chip. Thread criticality and scalability are two such approaches for parallel workloads. This section will review prior works on DVFS followed by research into methods of thread criticality prediction and scalability analysis.

2.1 Dynamic Voltage and Frequency Scaling on Chip Multi-Processors

During the mid-90's, the height of the "Megahertz War", processor manufacturers pursued increasing processor clock frequency in an attempt to gain performance improvements over rivals. This increase in clock frequencies, along with increased transistor density unfortunately lead to ever increasing power requirements as well. In an attempt to reign-in the ever increasing power envelopes of microprocessors, research began shifting away from single clocked global synchronous systems and toward globally asynchronous, locally synchronous systems (GALS)[19]. On system that uses GALS style clocking is Multiple Clock Domain (MCD) processors[31]. With this design, each functional block of a microprocessor is clocked independently of each other. This technique not only helped solve some of the clock routing issues that arose during the "Megahertz War", but also introduced new option of having each block being able to operate at independent voltage and frequency points. This design change allowed for the exploration of new dynamic voltage and frequency scaling techniques with the goal of improving performance and save energy during operation. As a result, researchers have been investigating a new problem of when and how to change voltage and frequency operating points.

The focus was initially on offline techniques, such as those discussed by Semeraro et al. in [31]. Utilizing a trace of a program, they developed a directed acyclic graph (DAG) to identify critical path and the paths that had "slack". They used this information to set operating frequency and voltage points to maximize performance and save energy. Magklis et al. [26] use profiling at compile time to inject reconfiguration instructions into the program to change the voltage/frequency levels while the program at runtime. This process necessitated multiple "training runs" of the program through the profiler in order to identify the optimal voltage/frequency settings during each phase.

While these offline techniques saved on hardware complexity, they ultimately were application specific. The programs would need to be profiled on various hardware configurations, and be compiled to include and hardware specific instructions for each every target platform. In contrast, an online method would allow for the program to be optimized, dynamically, even if it had never been run on that hardware configuration before. This was accomplished by looking at runtime characteristics in order to make determinations as to any voltage or frequency changes for the given hardware. In [30], the authors proposed using the processors queue utilization to develop an online algorithm to determine the optimal frequency. Their "Attack/Decay" algorithm leveraged the relation between the queue depth with the optimal frequency for that domain. The main idea of this paper was to look at the issue queues, and based on that information determine an appropriate frequency for each domain. If the queue was relatively unchanged, that indicated that the sender and receiver were operating at optimal frequencies and no change was needed. If the queue was increasing, the sender was performing operating faster than the receiver, and vice versa. In [36], the authors asserted that this methodology was heuristic-based, it was based on selected rules and specified threshold values, and as such, would be difficult to scale and improve upon according to the dynamic behavior that exists at runtime. Because of this, they proposed a analytical approach for modeling queues and domains. This model utilizes formulated linear equations relating to the demand and frequency of a given domain and use a Proportion-Integral-Derivative (PID) controller to solve them.

Grochowski et al. [14] decided to go a different direction, instead looking at the scalar and parallel phases of a program and design a microprocessor that can adapt. They determined that we can vary the energy expended on the execution of a process in relation to the parallelism that section exhibits. They formalized this relations as:

$$P = EPI * IPS \tag{2.1}$$

In this equation, EPI is the average energy spent on an instruction, IPS is average number of executed instructions per second and P is the fixed power budget. With a scalar section of code, the IPS will be low, indicating we can increase the EPI to improve performance without going over the power budget. In a similar vein, parallel code segments have a high IPS value, and we can spend less EPI to remain within power limits. This idea has lead to four different ideas to exploit this:

- Voltage frequency scaling
- Asymmetric cores
- Variable-size core
- Speculation control

This paper will look at an implementation the PCSLB algorithm[13] that uses DVFS to form pseudo-asymmetric cores in an attempt to show if, in the complexities of a real machine running a full operating system, the same level of performance and energy savings can be done. The method utilizes the idea of criticality to make artificial asymmetric cores using DVFS. This means that when in phases of lower parallelism, a large core is formed with high voltage and frequency, while in phases of low parallelism, a small core is formed by lowering voltage and frequency.

Isci et al. [18] introduced the concept of global power manager on chip. This concept looks at the power and performance of each core using feedback-control techniques. Through the evaluation of several different global dynamic power management policies, they show that these policies "perform better than static policies even if static scheduling is given oracular knowledge." Their policies included per-core DVFS to assign each core a independent power mode. Their best performing policy, MaxBIPS, works to find the optimal combination of DVFS for all core on the chip using an exhaustive search that predicts the power and Billion Instructions Per Second (BIPS) values for all possible combinations of DVFS levels. This method helps to identify the best possible application performance while remaining within the power budget. Due to the exponential nature of an exhaustive search, this method would work for a small number of cores, but as core counts begin increasing, this method is unscalable as projected in [25]. Instead, they propose a three-step scalable power control designed to handle both multithreaded and single threaded workloads simultaneously:

- First, the "aggregated frequency quota", a summation of the DVFS levels of all the cores normalized to the maximum DVFS level of a single core, is adjusted to maintain the chip within the power budget, resulting in a "chip level frequency quota".
- Next, all the cores running same applications are grouped together and the "chip level frequency quota" is divided among these groups.
- Lastly, the "group level frequency quota" is divided amongst the cores within a group according to the measured thread criticality.

2.2 Thread Criticality Prediction

Thread criticality has been a open research topic since at least the early 2000's. With multi-threaded programing, load workload imbalances between threads has been leads to energy inefficiencies during runtime. During execution, threads can share information between them, leading the need for synchronization methods to ensure only one thread can modify or utilize resources at a time. These synchronization methods, such as mutual exclusion (mutex) structures and barriers, lead to some threads becoming inactive, or not performing useful work, until it can "lock" access to these resources. These periods of inactivity waste energy as well as cause the threads to effectively run at different speeds, even if core frequencies are identical. The thread that runs the slowest, and thus provides an upper bounds to the programs performance as a whole, is termed the critical thread. Some of the first work in the area of identifying critical threads was done by Li, et. al., [23] which looked at the time taken by threads to reach "thrifty barriers" in a parallel program to calculate core frequencies. In their work, the last thread to reach this barrier was identified as the critical thread. Since the program could not continue execution until this critical thread reach the barrier, that meant they could safely reduce the frequency or even shutdown the cores for non-critical threads to conserve energy. In [23], these non-critical threads' cores were clock-gated and put into lower sleep modes, while in [24] these cores' frequencies were lowered using DVFS. While these techniques did show energy savings, the prediction of the critical thread was done using a "last-value predictor" which used the current barrier stall time to predict future times. This means that all threads have to run at the normal frequency until the barrier is reached. In [5], the authors attempt to resolve this by proposing to dynamically check running threads for imbalances at checkpoints they termed "meeting points." This was accomplished by testing with parallel loops with "meeting points" inserted at the end of the loops. These points allowed for the counting of loop iterations completions that are then used as predictor values for criticality prior to any threads reaching the barrier. With this technique, significant energy savings were achieved, but was limited to use on parallel loops. This lead to the generalize concept of thread criticality prediction by Bhattacharjee, et. al.,[3]. Here they proposed the use of different architectural parameters while can impact a thread's criticality. They found that cache misses was the most effective metric for predicting thread criticality due to the more cache misses a thread has, the slower it has to execute while waiting for loads from memory. They designed "criticality counters" to keep track of L1 and L2 cache misses for each thread, and, over each interval, these counters were used to determine the critical thread. While they did provide a good insight as to what parameters to consider, these cache based metrics could not be considered convincing since cache misses are mostly core frequency invariant. In a followup work, DuBois, et. al.,[9] looked at a different approach looking at a threads inactive periods due to blocking. In their work, they used an offline method of tracking periods of thread inactivity by looking at which threads are blocked and which are active over a given time interval and dividing that time interval amongst all active threads. This accumulated value was used as the metric for determining thread criticality. After running the program to collect this metric, the cores were then tuned prior to a second execution based on their criticality metric and the program was run again. While they did achieve a good offline estimation for thread-level load imbalance, this technique does not take into account the delays due to off-core accesses that result in a thread not being scalable and thus less sensitive to frequency changes.

2.3 Thread Scalability Prediction

Determining how a thread's performance will scale with change to a scaling voltage/frequency level is critical in order to energy efficient DVFS decisions. Increasing the frequency of thread, and the corresponding voltage level, to a thread that cannot take advantage of that frequency increase results in wasted energy, and the opposite can lead to a severe reduction in performance which can lead to additional energy inefficiencies due to the impacted load balance between threads. Over the years, there have been many works covering predicting the performance impact of frequency scaling. Benjamin and David [22] did an extensive study looking at various micro-architectural parameters which can impact and predict the performance for different applications. Through the use of regression models, they validated the strengths of different predictors. In contrast to this offline analysis, Kihwan Choi, et. al., [6] proposed a dynamic regression algorithm to model the scalability of a chip by looking at the ratio of off-chip access time to on-chip computation time. This method gave them a metric by which a thread can be deemed scalable or not. Another metric than can be utilized to determine the scalability of a thread is to look at the commit bandwidth[11]. These stalls due to being unable to commit instructions can be one cause of a thread not being scalable. These stalls can be related to outstanding memory instruction which are non-scalable due the off-core accesses which are frequency invariant. While they may be a cause, they cannot accurately predict as they may be independent instructions committing under an outstanding memory access. In [11] [21] [29], the off-chip latency of the first load miss in a series of load instructions are used to provide an abstract view of non-scalability periods. These latencies are accumulated to predict the non-scaling period of time. For Rustam, et. al., [27], the leading load assume a constant memory access time which can be inaccurate for real memories. Instead, the proposed the CRIT algorithm, which accumulates the variable latencies of load misses along the critical path of dependent memory instructions. All these techniques, though, have looked only at single threaded applications. In the very recent work [1], Akram, et. al., extended the CIRT algorithm, termed DEP+BURST, for use with multi-thread workloads taking inter-thread synchronization into account to determine system performance at different frequencies.

For PCSLB[13], Girdhar, developed a regression based scalability model similar to [22] for use with multi-threaded workloads. This work models scalability as a linear function of various architectural parameters and the the corresponding coefficients to represent their weights. Unlike in [22], Girdhar found that L1 Data and L2 cache misses were the most important parameters for predicting scalability of a thread. This work uses this linear model to determine a given thread's scalability value.

The PCSLB Algorithm proposed in [13] is given in Algorithm 1. The algorithm depends on calculating the lack of each thread relative to the critical thread, given by (3.1), then using that information to calculate a target time for a thread using (3.2). Once a targeted execution time is determined a frequency is determined that should result in that execution time using (3.3).

$$ExpectedGain_i = ExpectedGain_{critical} - Slack_i \tag{3.1}$$

$$ExpectedGain\% = \frac{T_{current} - T_{target}}{T_{current}} * 100 \tag{3.2}$$

$$T_{target} = T_{current}(S * (\frac{F_{current}}{F_{target}} - 1) + 1) \tag{3.3}$$

Algorithm 1 Proactive load balancing

```
Require: End of Control Period        ▷ Run the algorithm after every control period
 1: for i:1 to N do
 2:     $Slack_i \leftarrow \frac{C_{critical} - C_i}{C_{critical}} * 100$        ▷ Calculate Slack for each thread
 3:     $Scalability_{i_{current}} \leftarrow$ Calculate scalability for each thread using equation (1.2)
 4:     $Scalability_i \leftarrow \frac{Scalability_{i_{current}} + Scalability_{i-1}}{2}$        ▷ exponential moving average
 5: end for
 6: $T_{F_{max}}(critical) \leftarrow$ Predicted execution time for critical thread at $F_{max}$ using equation (3.3)
 7: $ExpectedGain_{critical} \leftarrow$ Predicted ExpectedGain for critical thread when run at $F_{max}$ using
    equation (3.2)
 8: for i:1 to N do
 9:     $ExpectedGain_i \leftarrow ExpectedGain_{critical} - Slack_i$        ▷ Calculate required ExpectedGain
    for each thread
10:     $T_{target_i} \leftarrow$ Calculate Target execution time based on $ExpectedGain_i$ using equation (3.2)
11:     $F_{target_i} \leftarrow$ Calculate Target frequency using $T_{target_i}$ from equation (3.3)
12: end for
```

This algorithm forms the basis of this work. It has been implemented in three stages, first with a custom kernel module, second by modifying the *futex_wait* routine, and finally my modifying the *intel_pstate* driver.

3.1 Creation of CritScale Module

The main core the work is the implementation of the PCSLB[13]. To accomplish this task a custom kernel module was created called CritScale. This module was designed as a built-in module due to it's interaction with other modules within the Linux kernel. The main thread is designed following the PCSLB algorithm. After creation, the thread will sleep for a specified interval, by default 10 ms, then wake and check if there is a program registered with the module to be tracking for DVFS calculations. If no processes is registered, or the previously registered process has exited, the thread will then skip to the end of the main loop and return to sleep. If a process is registered and active, the thread will then proceed to with the following steps:

- First, update the criticality for all threads.
- Second, update scalability values for all threads.
- Third, determine the critical thread by comparing criticality values for all threads.
- Fourth, assign the requested frequency for the core running critical thread to F_{max}, and calculate the requested frequencies for all other threads.
- Fifth, if multiple threads are running on the same core, set the requested frequency to the maximum value.
- Sixth, bound all frequencies to a range between F_{max}, and F_{min} for the cores.
- Finally, write the the IA32_PERF_CTL register of the CPU the value stored as the requested frequency.

For tracking Criticality of a thread, the idea of epochs are used as standard update periods. These epochs begin and end whenever a thread in the thread group unable to acquire a lock and is blocked, or acquires a lock and is unblocked. The time of these epoch is then divided equally amongst all thread active during the epoch and added to the criticality accumulator in the threads' task_struct structure. After all the threads have had their criticality updated, the thread that is transitioning is then flagged as active or inactive accordingly for the next epoch. This method is derived from the proposed online criticality counters proposed by [9]. This method uses Equation 3.4 for updating criticality at the end of each epoch, where T_i is the length of the interval in time, and N is the number of active threads during the epoch.

$$Criticality_{i_{current}} = \begin{cases} \frac{T_i}{N} & \text{, if } Active_i \\ 0 & \text{, Otherwise} \end{cases} \tag{3.4}$$

As identified by [13], tracking of L1D and L2 cache misses are important for determining the scalability of a thread. To implement this, the *perf_events* subsystem[34] was utilized. This system call API allows for interaction with the core's dedicated performance event counters. Whenever a thread in the tracked thread group is created, the *scalability_attach* routine creates performance counters for the retired instructions, L1D cache misses, and L2 cache misses. These events are then attached to the thread and are set to update regardless of which core the process is running on so as to make sure the performance counters follow the thread through any scheduler migrations. When the new value for the scalability is needed, these performance counters are read and the difference between the current values and the previous values are calculated. These new values represent are then using the algorithm [INSERT ALGORITHM]. This scalability value represents the scalability of the last evaluation interval. This new scalability and the old scalability algorithm is then used to calculate the running average of scalability.

The requested frequencies for each thread is calculated using (3.2) and (3.3) from [13]. Over the course of implementation, in order to reduce the number of calculations required to determine the requested frequency, the formulas (3.2) and (3.3) were combined and reduced to a single equation (3.5). This equation also shows that the requested frequency is not dependent on the time interval for the evaluation.

$$F_{target} = F_{current} * \frac{S}{S - ExpectedGain} \tag{3.5}$$

With this simplified equation, the updated full algorithm for the main thread is given by 2. This time to execute this algorithm will scale linearly with the number of thread in the thread group. This is due to the O(n) complexity of each loop.

3.2 Modification of Futex

In order to track if a thread is blocked, modification of the *futex_wait* routine is needed. All the benchmarks for evaluation utilizes the pthread library. In the library, the *pthread_mutex_lock* and *pthread_mutex_unlock* call the *futex_wait* and *futex_wake* system-calls, respectively. Since the *futex_wait* routine will determine if a thread is blocked or not, it was modified. If the thread is blocked, it will enqueue the thread on the *wait_queue* which will indicate to the scheduler to pre-empt it and unschedule it from the core. After another process calls *futex_wake* as part of unlocking, the scheduler will then wake the thread and re-schedule in onto a core. In order to accurately track the blocking epochs for criticality calculations, the *sys_deactivate* routine is called upon entering *futex_wait* to update the criticality of the thread group. After the thread is woken up by an unlock, the thread will resume execution inside *futex_wait* where it will be verified that the wake signal was valid. If it is a valid wake signal, the *sys_activate* routine is called to update criticality of the thread group before returning to normal execution.

3.3 Modification of Intel P-State Driver

In order to take advantage of the PCPS feature of Haswell-EP, we need to utilize the *intel_pstate* driver. This driver will configure the microprocessor's power management system to coordinate with the driver to set frequencies. The only modification within the driver was removing the code to set frequencies. This leaves our module as the only module in the kernel that can set frequencies to the core, ensuring what frequencies we set are the frequencies the cores receive.

Algorithm 2 Main thread using proactive load balancing

```
Require: End of Control Period and Registered Thread Group  ▷ Run the algorithm after every control period
 1: for i:1 to M do
 2:     $Rq\_Freq_i \leftarrow F_{min}$    ▷ Set initial Requested Frequency for each core
 3: end for
 4: for i:1 to N do
 5:     if $i_{Active}$ then    ▷ Only update for active threads
 6:         $Criticality_i \leftarrow Criticality_{i_{lastperiod}} + \frac{TimeDiff}{NumActive}$    ▷ Updated criticality based on 3.4
 7:     else
 8:         $Criticality_i \leftarrow Criticality_{i_{lastperiod}}$
 9:     end if
10:     $Criticality_{i_{lastperiod}} \leftarrow 0$    ▷ Reset accumated criticality for next sampling interval
11: end for
12: for i:1 to N do
13:     if $Criticality_i > Criticality_{Critical\_thread}$ then
14:         $Critical\_Thread \leftarrow Criticality_i$
15:     end if
16: end for
17: for i:1 to N do
18:     $Slack_i \leftarrow \frac{C_{critical} - C_i}{C_{critical}} * 1000$    ▷ Calculate Slack for each thread
19:     $Scalability_{i_{current}} \leftarrow$ Calculate scalability for each thread using equation (1.2)
20:     $Scalability_i \leftarrow \frac{Scalability_{i_{current}} + Scalability_i}{2}$    ▷ Exponential moving average
21: end for
22: $T_{F_{max}}(critical) \leftarrow$ Predicted execution time for critical thread at $F_{max}$ using equation (3.3)
23: $ExpectedGain_{critical} \leftarrow$ Predicted ExpectedGain for critical thread when run at $F_{max}$ using equation (3.2)
24: for i:1 to N do
25:     $ExpectedGain_i \leftarrow ExpectedGain_{critical} - Slack_i$    ▷ Calculate required ExpectedGain for each thread
26:     $F_{target_i} \leftarrow$ Calculate Target frequency using $ExpectedGain_i$ and $Scalability_i$ from equation (3.5)
27:     if $F_{target_i} > Rq\_Freq_{i_{cpu}}$ then
28:         $Rq\_Freq_{i_{cpu}} \leftarrow F_{target_i}$    ▷ Ensures that the maximum requested frequency is set
29:     end if
30: end for
31: for i:1 to M do
32:     $CPU_{m_{Freq}} \leftarrow Rq\_Freq_m$    ▷ Set the CPU Frequency to the Requested Frequency
33:     $F_{Current_m} \leftarrow Rq\_Frq_m$    ▷ Update Current Frequency data structure
34: end for
```

4. IMPROVEMENTS TO THE PCSLB ALGORITHM

4.1 Frequency Scaling

The PCSLB algorithm provides many insights into utilizing thread criticality, scalability, and the concept of slack for a DVFS scheme. Through testing though, the issues became apparent in it's implementation.

First, due to the equations used to determine at target frequency for non-critical threads, there were frequent cases where the thread was set to a target frequency of the minimum frequency allowed by the processor. When plotting (3.5) as a function, Figure 4.1, of the non-critical threads expected gain, with fixed current frequency and scalability, we see a vertical asymptote around where expected gain equals scalability. As we approach this asymptote from the left, we scale towards positive infinity, while approaching from the right scales towards negative infinity. This asymptote indicates that as expected gain approaches scalability, the core frequency required goes outside the available frequencies for that core. What lies beyond this barrier is information that doesn't carry any meaning, and due to the extremely negative values that are output by the equation, the processor is set to it's minimum frequency.

This is exhibited in *blackscholes* most clearly. As shown in Figure 4.2, the critical thread is promoted to running at maximum frequency while all other threads run at minimum frequency despite the fact that from sampling to sampling, the slack remains low. This initially occurs due to a large expected gain for the critical thread (Steps 19 and 20 in Algorithm 2), in combination with the low slack of the non critical threads. When this occurs, the threads have all started to processes data on new cores, resulting in high miss rates and lowering their scalability values. As a result, the critical thread is promoted to the maximum frequency, but all non-critical threads enter the highly negative region of the graph because expected gain is greater than the current scalability.

The included heatmaps can be read as each column representing a core, and rows representing a sampling interval with time progressing from top to bottom. The coloring ranges from green for

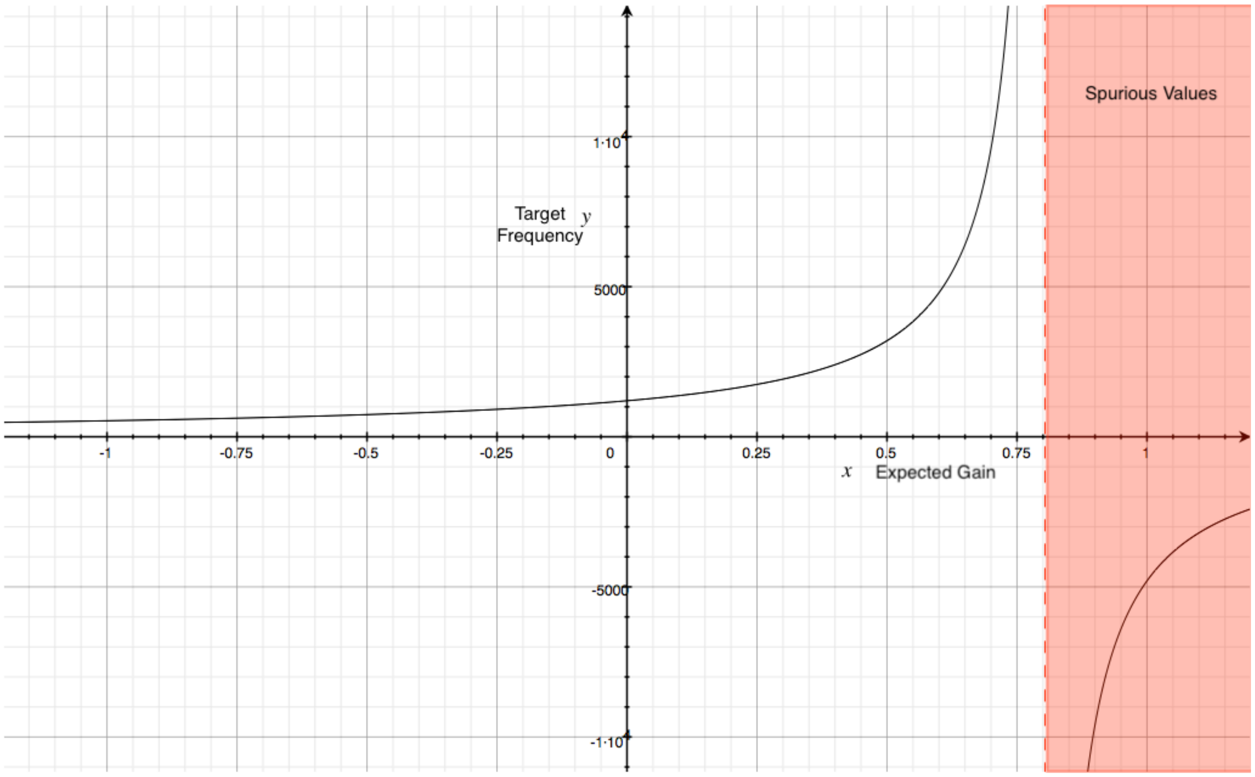


Figure 4.1: Target Frequency vs Expected Gain for $F_{current} = 1200MHz$ and $Scalability = 0.8$

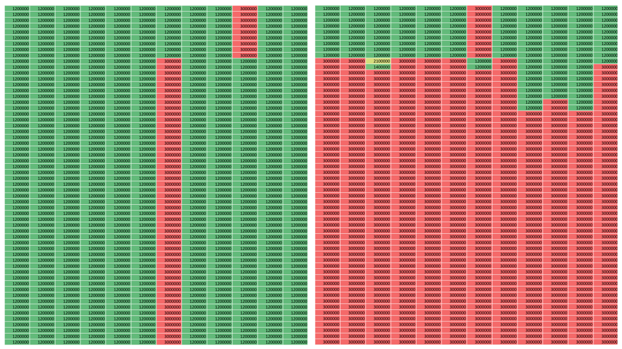


Figure 4.2: Core Frequencies running *blackscholes* using (3.5) (left) and (4.3) (right)

low frequency cores to red for high frequency cores.

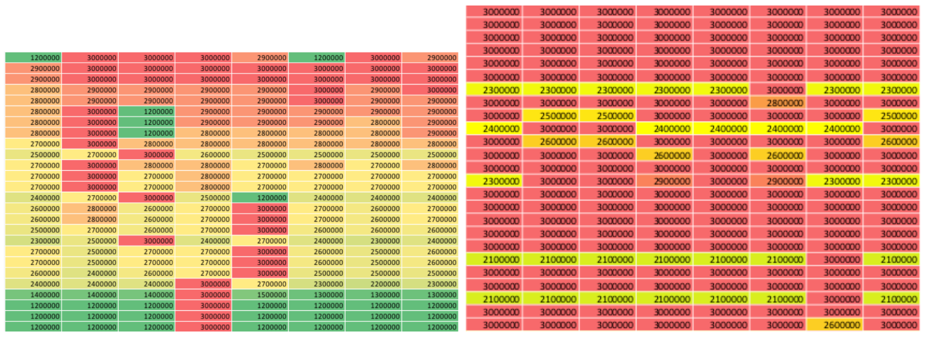
Additionally, the system is not given a chance to recover from this spurious result. In this case, since the critical thread remains constant throughout the multi-threaded portion of the application, the expected gain from steps 19 and 20 in Algorithm 2 is zero. Since there is very little blocking between threads in *blackscholes*, the slack remains constantly low, if not zero. Because of this, for a non-critical thread, the expected gain is zero, resulting in (3.5) outputting the target frequency is the current frequency (the incorrect minimum frequency).

Another issue noted is the equations used are over-eager to continuously decrease the frequency of a core if we are continuously correctly selecting the critical thread each interval. In these cases, the resulting expected gain for the critical thread is zero, and for non-critical threads is $-slack$. Because of this, we continue to decrease the frequency of the cores for non-critical threads, eventually resulting in them all running at or close to F_{min} for extended periods of time. This can be shown in the heat-map of *facesim* and resulting in loss of performance, Figure 4.3.

4.2 Modification of PCSLB

With the issues noted, we can look at ways to resolve them. For the issue with the spurious region where expected gain is greater than scalability, a piecewise element was added to the equation. The purpose of this was to ensure that if we wandered into this region, the results would still carry meaning. Since we're cut off where $exp\_gain \geq scalability$ since it results in a frequency that is not achievable, we will select a frequency that is, F_{max}.

In order to counter act the continuous push towards minimum frequencies, (3.5) will need to be modified. Since the slack is a measure of load imbalance due to criticality, scaled to the criticality of the critical thread, the frequencies selected should reflect that performance relative to the critical thread. With this in mind that scaling of non-critical threads frequencies should be based on the critical thread's frequency, F_{max}. This is derived by replacing $T_{current}$ with $T_{F_{max}}$ of the critical thread in (3.2) and (3.3), and $F_{current}$ with F_{max} in (3.3).



1200000	3000000	3000000	3000000	2900000	1200000	3000000	2900000
2900000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2900000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2800000	2900000	2900000	2900000	2900000	3000000	2900000	3000000
2800000	2900000	2900000	2900000	2900000	3000000	2900000	2900000
2800000	3000000	1200000	2900000	2900000	2900000	2900000	2900000
2800000	3000000	1200000	2900000	2900000	2900000	2800000	2900000
2800000	3000000	1200000	2800000	2800000	2800000	2800000	2900000
2700000	3000000	2800000	2800000	2800000	2800000	2800000	2800000
2500000	2700000	3000000	2600000	2500000	2500000	2500000	2500000
2700000	3000000	2800000	2800000	2700000	2800000	2700000	2800000
2700000	3000000	2700000	2800000	2700000	2700000	2700000	2700000
2700000	3000000	2700000	2800000	2700000	2700000	2700000	2700000
2400000	2700000	3000000	2500000	1200000	2400000	2400000	2400000
2600000	2800000	2600000	2700000	3000000	2700000	2600000	2700000
2600000	2800000	2600000	2700000	3000000	2700000	2600000	2700000
2500000	2700000	2600000	2700000	3000000	2600000	2600000	2600000
2300000	2500000	3000000	2400000	2700000	2400000	2300000	2400000
2700000	2500000	2700000	2700000	3000000	2600000	2500000	2600000
2700000	2500000	2700000	2700000	3000000	2600000	2500000	2600000
2600000	2400000	2600000	2700000	3000000	2500000	2500000	2500000
2400000	2400000	2400000	3000000	2700000	2300000	2200000	2300000
1400000	1400000	1400000	3000000	1500000	1300000	1300000	1300000
1200000	1200000	1200000	3000000	1200000	1200000	1200000	1200000
1200000	1200000	1200000	3000000	1200000	1200000	1200000	1200000
1200000	1200000	1200000	3000000	1200000	1200000	1200000	1200000

3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2300000	2300000	2300000	2300000	2300000	3000000	2300000	2300000
3000000	3000000	3000000	3000000	3000000	2800000	3000000	3000000
3000000	2500000	2500000	3000000	3000000	3000000	3000000	2500000
2400000	3000000	3000000	2400000	2400000	2400000	2400000	3000000
3000000	2600000	2600000	3000000	3000000	3000000	3000000	2600000
3000000	3000000	3000000	2600000	3000000	2600000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2300000	3000000	3000000	2900000	3000000	2900000	2300000	2300000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2100000	2100000	2100000	2100000	2100000	2100000	3000000	2100000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
2100000	2100000	2100000	2100000	2100000	2100000	3000000	2100000
3000000	3000000	3000000	3000000	3000000	3000000	3000000	3000000
3000000	3000000	3000000	3000000	3000000	3000000	2600000	3000000

Figure 4.3: Core Frequencies running *facesim* using Equation 3.5 (left) and Equation 4.3 (right)

$$ExpectedGain\% = \frac{T_{F_{max}} - T_{target}}{T_{F_{max}}} * 100 \tag{4.1}$$

$$T_{target} = T_{F_{max}}(S * (\frac{F_{max}}{F_{target}} - 1) + 1) \tag{4.2}$$

Thus, with this two modifications we arrive at the equations (4.1) and (4.2) which then simplifies to (4.3) which replaces (3.5) in Algorithm 2.

$$F_{target} = \begin{cases} F_{max} & \text{, if } ExpGain \geq Scalability \\ F_{max} * \frac{Scalability}{Scalability - ExpGain} & \text{, otherwise} \end{cases} \tag{4.3}$$

With this equation incorporated, we see improvements in both cases described above, as shown by Figures 4.2 and 4.3.

4.3 Criticality History

Both [9], in their dynamic version, and [13] calculate criticality solely over a given sampling interval and clear the counters for the next interval. This form of (almost) memoryless calculation bases the decision on frequencies for the next interval solely based on information gathered in the previous interval. As can be seen in Fig. 4.4, this value can he highly variable, and as such lead to drastic shift in frequencies the cores are set to. As such, memoryless calculations can result in inefficiencies and possibly introduce additional slack as a result of some cores being scaled back

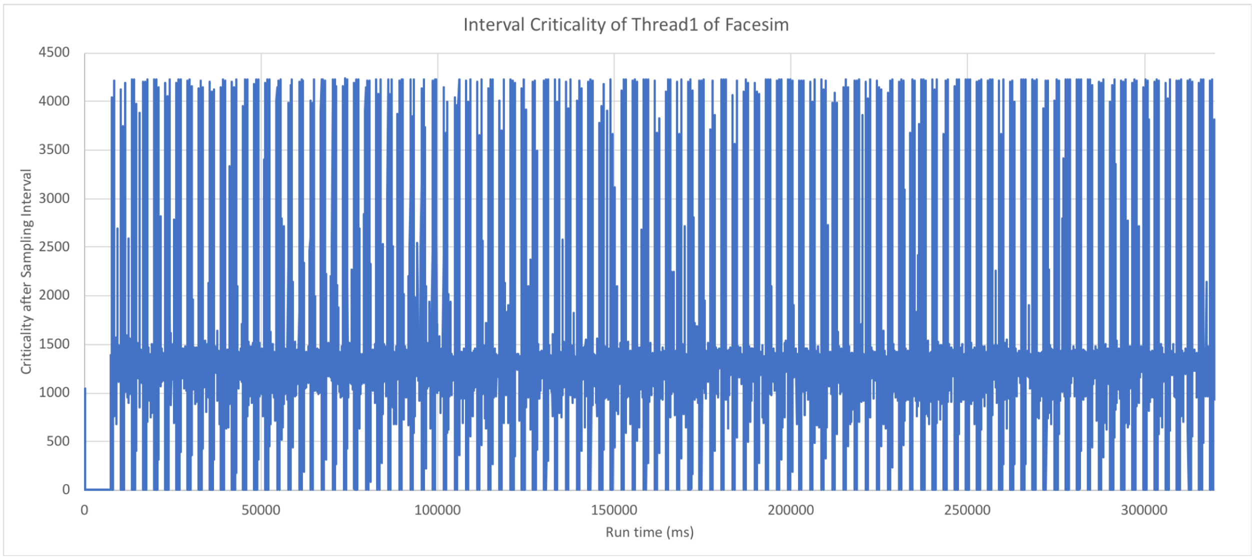


Figure 4.4: Value of Criticality during each Sampling Interval

too far for the next decision interval.

One way of resolving the high variability of criticality is to include some degree of history into the criticality metric. The use of an exponential moving average would allow for "smoothing" of short lived spikes and also allowing for sustained periods of high criticality to be carried through.

To incorporate the history, whenever the main thread updates criticality, rather than reading the accumulated value and then clearing the counters, instead the accumulated value for that decision interval is averaged using Equation 4.4. In this way, the current criticality is averaged with the historical criticality. The weights for each value can be tuned in order to identify the best performance for the application that is being tracked.

$$Criticality_{i_{historical}} = \frac{W_1 * Criticality_{i_{historical}} + W_2 * Criticality_{i_{current}}}{W_1 + W_2} \tag{4.4}$$

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

The machine we are using for all tests is a Dell workstation with the configuration given in Table 5.1.

An Intel Xeon E5-2687W v4 was selected due to the energy features of the Haswell/Broadwell-EP architecture, mainly the ability to set different P-States for each core independent of the other cores' settings. This ensured each core was operating at the specified voltage and frequency point as requested by the CritScale module. Additionally, the 12 core setup allows us to execute up to 12 threads on separate cores, helping to reduce the thread contention for execution time on a single core.

The kernel is configured to use the modified *intel_pstate* driver without hardware coordination to ensure the only component making decisions on the target P-State for each core is the CritScale module.

5.1.1 Energy Consumption Collection

Due to the complexities of measuring the power consumption for a CPU[15], there is no definitive method for collection of CPU power metrics. The two commonly accepted way are through the Intel Running Average Power Limit (RAPL) system built into modern Xeon processors, or mea-

ISA	x86-64
# of Cores	12
Core Type	4-wide, out-of-order
L1D Cache Size	384KiB
L2 Cache Size	3 MiB
Frequency Range	1.2 - 3.0 GHz
Voltage Points	0.78 - 1.82 V
# of P-States	19

Table 5.1: Specifications for Intel Xeon E5-2687W v4

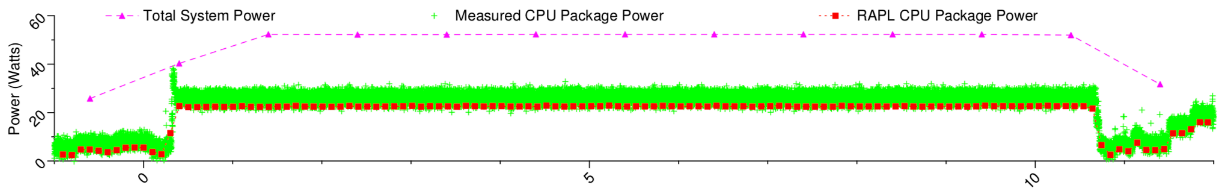


Figure 5.1: Comparison of RAPL values vs. Measured values [8]

suring power consumption at points prior to the CPU's voltage regulators (VRs)[17]. This method though, runs the risk of over measuring CPU power due to inefficiencies in the VRs. The RAPL system has been shown to be consistent in its readings, but with a small fixed offset from power measured at the VRs[8], most likely due to the difference in energy from the VRs inefficiency. The effects of this offset are shown in Figure 5.1. For our workstation setup, there are a total of 24 +12V lines for CPU power to both sockets. Due to the complexities of configuring 12 hall-effect sensors to be spliced into the +12V lines for the CPU, difficulties in identifying which lines power which sockets, and possibility of over measuring CPU power prior to the VRs, the simplicity of the RAPL system allowed for quick integration of power measurements.

5.1.2 Benchmarks

The PARSEC benchmark suite [4] was selected to the large number of types of benchmarks available. These benchmarks have a high variability in scalability and inter-thread communication allowing them to test the module under a large selection of workload types. Additionally, the suite includes the SPLASH-2 benchmarks [35]. The benchmarks generally fall into two categories for how they utilize threads. Some are data parallel where the threads operate on independent chunks of data with little communication between threads; providing the best-case scenario for the module. The others are "data pipeline" where each thread will perform different actions on the same chunk of data, similar to a pipelined processor, resulting in a high degree of inter-thread communications and the worst-case scenario for our module. Table 5.2 outlines the benchmarks used and their type.

In order to indicate to the module which thread group to track, a syscall (*sys_track_me*) was implemented in the CritScale module to get a pointer of the current *task_struct* of the calling

Benchmark	Parallel Model	Parallel Granularity	Thread Communication
blackscholes	Parallel	Coarse	LOW
bodytrack	Parallel	Medium	HIGH
canneal	Unstructured	Fine	HIGH
facesim	Parallel	Coarse	MEDIUM
fluidanimate	Parallel	Fine	MEDIUM
ocean_ncp	Parallel		LOW
lu_ncb	Parallel		HIGH

Table 5.2: PARSEC Benchmarks

process and set the *track_task* to point to it. This syscall was then inserted as the first function the benchmarks will execute in their *main* function to ensure we begin tracking as early as possible.

5.2 Evaluation

All results presented here are the geometric means of 10 runs of the given benchmark for the labeled method. This was done to ensure variation due to outside influences such as other processes sharing the system was minimized.

5.2.1 Results: PCSLB vs PCSLB-Max

In this section we will examine the impact of the improvements in frequency calculations made to the PCSLB algorithm (termed PCSLB-Max). Both algorithms use a accumulative counter for criticality as we will look at the impact of using various levels of history in the next section.

As expected, the initial results from the PCSLB algorithm are far from expected values. In [13], *bodytrack* was demonstrated to consume 2% less energy while only suffering from a 2% performance loss. When integrated as a full DVFS scheme on a real machine, it did consume 11% less energy, but was taking 11% longer to execute. Similar metrics can be seen from all the benchmarks, with *facesim*, *fluidanimate*, and *blackscholes* actually consuming more energy due to the drastically increased execution time. As stated in Chapter 4, this is due to how the PCSLB algorithm scales back core frequencies to the minimum quite frequently. As a result, nearly all cores run at the minimum frequency of 1.2GHz while only the critical thread runs at anything

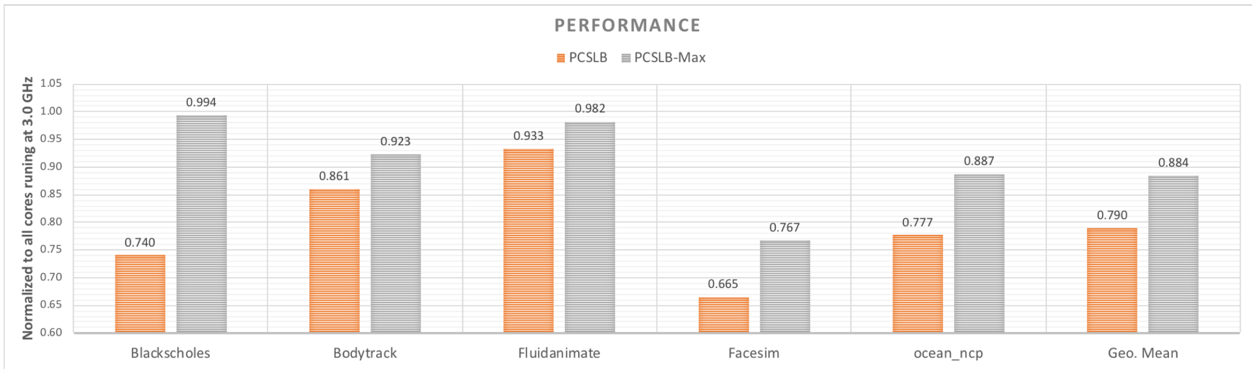


Figure 5.2: Performance Comparison of PCSLB to PCSLB-Max

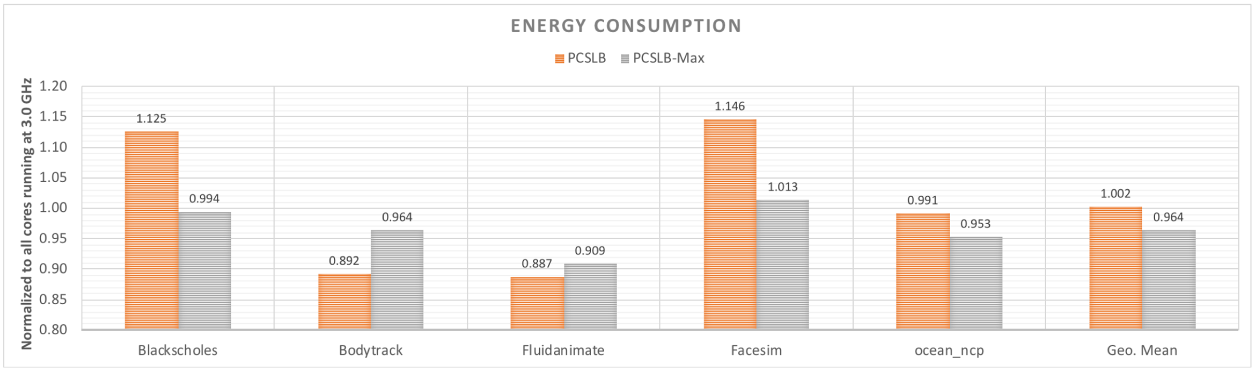


Figure 5.3: Energy Comparison of PCSLB to PCSLB-Max

higher as shown in Figures 4.3 and 4.2. The impact to *blackscholes* is particularly important due to it's embarrassingly parallel nature and near perfect thread balance due to no inter-thread communication. Ideally, any DVFS method applied to *blackscholes* will not negatively impact it.

When applying the modifications to PCSLB's equations for finding the target frequency for non-critical threads outlined in Chapter 4, the performance of each benchmark improves, but still is able to show energy savings. With *fluidanimate*, the performance is within 1% while still managing to save almost 12% in energy consumption. Most benchmarks show between 2-5% performance degradation and 1-2% energy savings though, showing more room for refinement.

Graph label	W_1	W_2
(0 - 100)	0	100
(25 - 75)	25	75
(33 - 66)	33	66
(50 - 50)	50	50
(66 - 33)	66	33
(75 - 25)	75	25

Table 5.3: Weights Chosen for Exponential Moving Average

5.2.2 Results: Effects of History Weights

Next we will look at the the effect of various weights on the effectiveness of the PCSLB-Max algorithm. The weights chosen for analysis are given by Table 5.3. These weight will bias the calculations in favor of either the historical values or the current value for criticality. Additionally, the results from using a pure accumulation of criticality, as used by [9] in their offline metric, is included as well.

As we can see from Fig. 5.4, there are some benchmarks where the effect of the weights does impact performance of the benchmark, such as *facesim* and *fluidanimate*. In this case, the more we weight to the historical value, the worse the performance impact. Again, *blackscholes* shows up as a cornerstone benchmark showing very little variation due to various weights. Additionally, *bodytrack* shows very little change in performance due to the change in weights. While there is a variation, all values are within 1% of each other, meaning the variation is likely due to outside causes such as other processes on the system. On average we see that generally, there is a best weight for the benchmarks in the around the (25 - 75) point. From Fig. 5.5, for all cases though, we still see an energy savings that are relatively stable, within 3% of each other. Since we aiming to minimize performance impact while reducing energy consumption, we should also look at the trade off as "Performance/Joule" in Fig. 5.6. In this metric, the more we are greater than one, the more we are saving in energy than we are losing in performance. In this case, we can see that for some benchmarks, such as *lu_ncb* and *blackscholes*, the energy savings scale linearly with

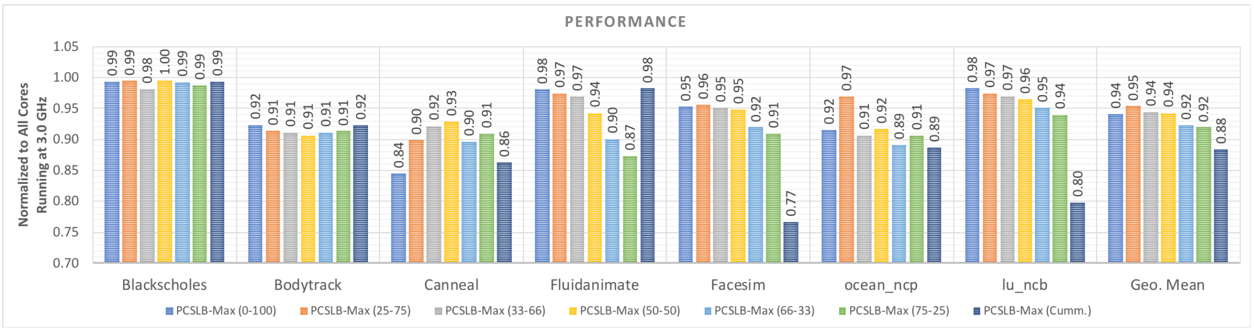


Figure 5.4: Performance Comparison of Different History Weights for PCSLB-Max

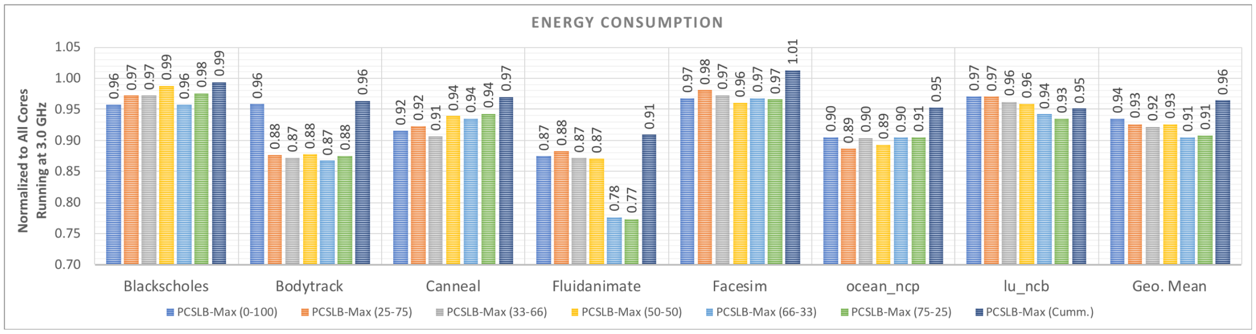


Figure 5.5: Energy Consumption Comparison of Different History Weights for PCSLB-Max

performance loss, indicating this technique might not be optimal for these sorts of applications. *Canneal* stands out though as the worst case since in nearly all variants, our performance loss is greater than the energy saving. Most applications though are good targets for this technique due to non-linear energy savings with respect to any performance loss.

Looking at the mean of all the results, we can take the 25/75 weighting to provide the optimal results of the variants we have tested.

5.2.3 Results: PCSLB-Max vs CPUFeq Governors

Now that we have determined the best history weighting system for PCSLB-Max, we now compare it's results to two commonly used frequency governors using the CPUFreq interface, SchedUtil and OnDemand. OnDemand, introduced in 2006, is the simplest of the governors since it will dynamically adjust core frequencies based on CPU load. SchedUtil, in addition to being more

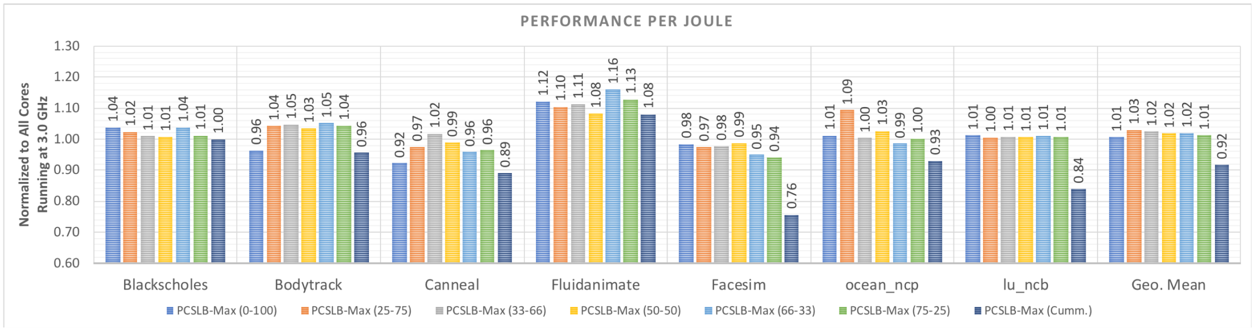


Figure 5.6: Performance per Joule Comparison of Different History Weights for PCSLB-Max

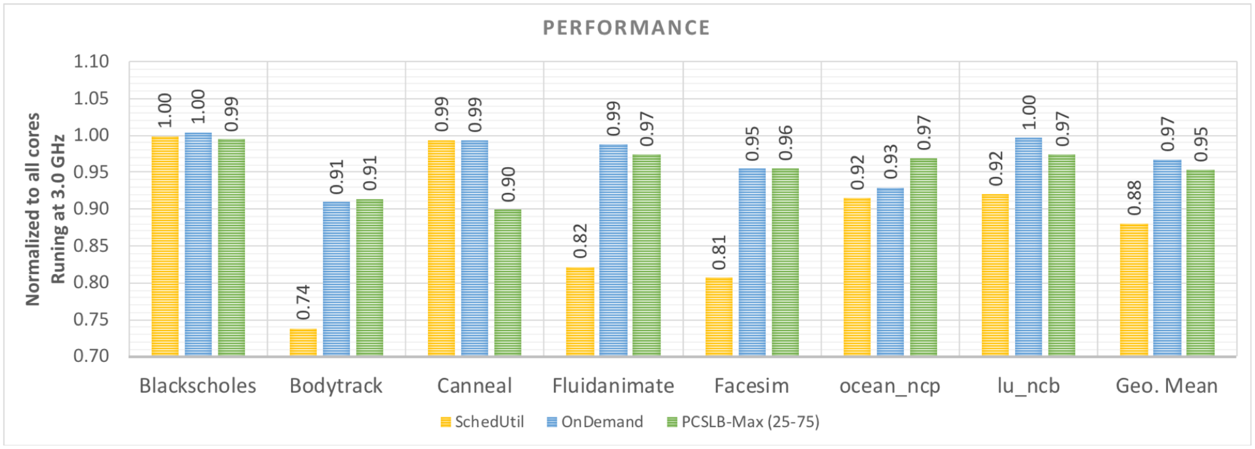


Figure 5.7: Performance Comparison of PCSLB-Max to CPUFreq Governors

recent (introduced in 2016), is more complicated, adjusting core frequencies based on scheduler load based on the Per-Entity Load Tracking (PELT) mechanism that is part the Linux scheduler.

Looking at performance in Fig. 5.7, we are able to outperform SchedUtil in nearly all cases, except *canneal*, with results that nearly meet, or in some cases exceed, those of OnDemand. The notable exception to this is *canneal* where we actually have noticeably worse performance than either.

Looking at energy consumption in Fig. 5.8, we are able to achieve more energy saving in nearly all cases than SchedUtil and OnDemand. Again though, we have exceptions this time with *fluidanimate* and *facesim* where we beat OnDemand, but lose out to SchedUtil, but in these cases we were still able to beat SchedUtil in performance by a large margin.

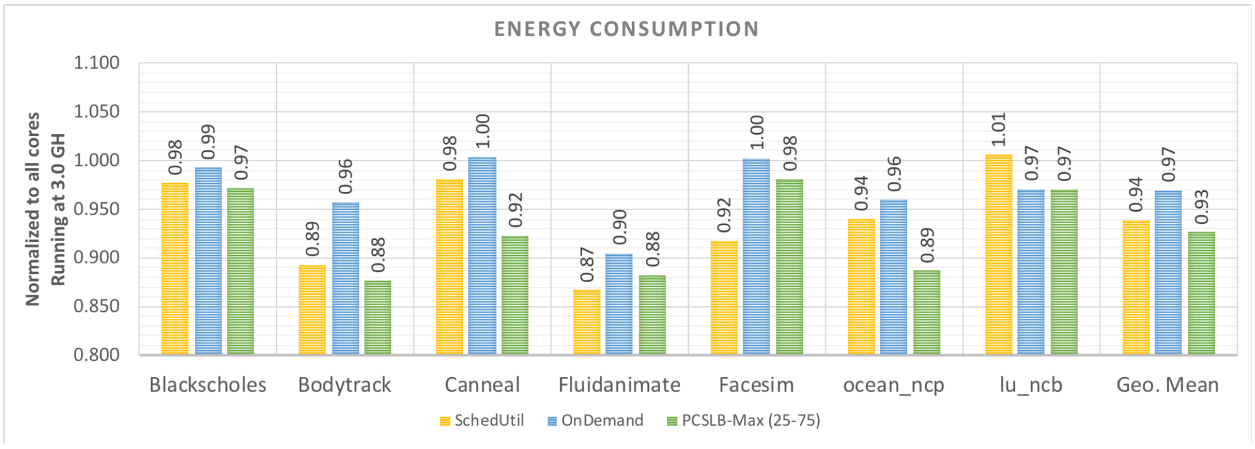


Figure 5.8: Energy Consumption Comparison of PCSLB-Max to CPUFreq Governors

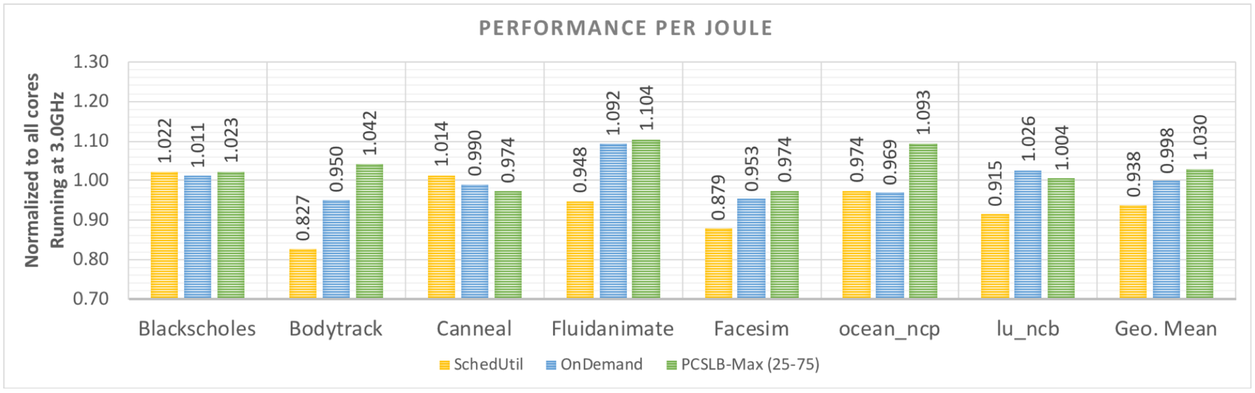


Figure 5.9: Performance per Joule Comparison of PCSLB-Max to CPUFreq Governors

Again, since we aim to minimize performance impact while maximizing energy savings, we turn towards the metric of "Performance per Joule" in Fig. 5.9 where we see we are able to increase our energy savings with less of a performance impact, on average, than both SchedUtil and OnDemand. Again though we have the notable exception of *canneal*, where SchedUtil was the best performing scheme for this benchmark.

6. CONCLUSION AND FUTURE WORK

Thread imbalance is inevitable due to the necessity of synchronization mechanics to coordinate threads. But this imbalance can can also be a useful tool to leverage for energy savings. As shown in this work, using the criticality metric as define by [9] can lead to real world energy savings and while minimizing performance impact. With this metric, and the slack metric defined by [13] be can begin to Despite initial reservations about the effectiveness of the PCSLB algorithm to achieve energy savings on a real machine, with the PCSLSB-Max algorithm we were able to achieve energy saving greater than the two commonly used Linux CPUFreq governors, while achieving similar performance to OnDemand, and actually beating SchedUtil. There will always be a trade off between energy and performance, but with an understanding of this thread imbalance of an application, we can can make smart decisions so as to use that imbalance to our advantage.

Looking towards the future, there are still more areas of opportunity for PCSLB-Max. As shown in the results, no history model will fit all programs, and as such the algorithm could be further refined though investigation of identifying ways to make the weights dynamic, allowing for the algorithm to adjust the weights to achieve optimal performance of the algorithm. As the criticality model stands now, all locks are treated the same, so it is difficult to distinguish whether, for example, two different threads are blocked by the same lock or by different locks. As such additional work can be done on the model of criticality to identify with thread is specifically blocking any given thread, allowing for more fine grained understanding of how blocking is causing thread imbalance.

REFERENCES

[1] S. Akram, J. B. Sartor, and L. Eeckhout. Dvfs performance prediction for managed multi-threaded applications. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 12–23, April 2016.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 298–309, June 2005.

[3] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. *SIGARCH Comput. Archit. News*, 37(3):290–301, June 2009.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 240–249, New York, NY, USA, 2008. ACM.

[6] Kihwan Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 4–9 Vol.1, Feb 2004.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[8] Spencer Desrochers, Chad Paradis, and Vincent M Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, pages 455–470. ACM, 2016.

[9] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. *SIGARCH Comput. Archit. News*, 41(3):511–522, June 2013.

[10] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 362–373, New York, NY, USA, 2011. ACM.

[11] S. Eyerman and L. Eeckhout. A counter architecture for online dvfs profitability estimation. *IEEE Transactions on Computers*, 59(11):1576–1583, Nov 2010.

[12] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, volume 85. AUUG, Inc., 2002.

[13] Nimish Girdhar. Predicting Thread Criticality and Frequency Scalability for Active Load Balancing in Shared Memory Multithreaded Applications. Master's thesis, Texas A&M University, College Station, TX, 2016.

[14] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *Proceedings of the IEEE International Conference on Computer Design*, ICCD '04, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[15] D. Hackenberg, R. Schne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904, May 2015.

[16] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, Oct 1994.

[17] T. Ilsche, D. Hackenberg, S. Graul, R. Schne, and J. Schuchart. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Dec 2015.

[18] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[19] A Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 158–168, 2002.

[20] D. Kadjo, U. Ogras, R. Ayoub, M. Kishinevsky, and P. Gratz. Towards platform level power management in mobile systems. In *2014 27th IEEE International System-on-Chip Conference (SOCC)*, pages 146–151, Sept 2014.

[21] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 287–296, New York, NY, USA, 2010. ACM.

[22] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGPLAN Not.*, 41(11):185–194, October 2006.

[23] J. Li, J.F. Martinez, and M.C. Huang. The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors. In *Software, IEE Proceedings-*, pages 14–23, Feb 2004.

[24] Chun Liu, Anand Sivasubramaniam, M. Kandemir, and M.J. Irwin. Exploiting barriers to optimize power consumption of cmps. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 5a–5a, April 2005.

[25] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. *SIGARCH Comput. Archit. News*, 39(3):449–460, June 2011.

[26] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *SIGARCH Comput. Archit. News*, 31(2):14–27, May 2003.

[27] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[28] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231, June 1985.

[29] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[30] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[31] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 29–, Washington, DC, USA, 2002. IEEE Computer Society.

[32] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[33] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[34] R Vitillo. Performance tools developments. *Future computing in particle physics*, 2011.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.

[36] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGOPS Oper. Syst. Rev.*, 38(5):248–259, October 2004.