PREDICTING THREAD CRITICALITY AND FREQUENCY SCALABILITY FOR ACTIVE LOAD BALANCING IN SHARED MEMORY MULTITHREADED APPLICATIONS

A Thesis

by

NIMISH GIRDHAR

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

August 2016

Major Subject: Computer Engineering

ABSTRACT

With advancements in process technologies, manufacturers are able to pack many processor cores on a single chip. Consequently, there is a paradigm shift from single processor to single chip multiprocessors (CMP). As mobile industry is moving towards CMPs, the challenge of working in a low power/energy budget while still delivering good performance, has taken the precedence. Therefore, energy and power optimization has become a primary design constraint along with performance in CMP design space. CMPs get performance benefit from running multithreaded programs which utilizes Thread Level Parallelism(TLP). But these parallel programs have inherent load imbalance between threads due to synchronization which degrades performance as well as energy consumption. The solution is to develop energy efficient algorithms which can detect and reduce this imbalance to maximize performance while still giving energy savings. Dynamic thread criticality prediction is an approach to detect the load imbalance by identifying the critical threads which cause performance/energy degradation due to synchronization. Cores running critical threads can then be run on high power states using Dynamic Voltage Frequency Scaling, thus balancing the execution times of all thread. But in order to create an accurate balance and utilize DVFS efficiently, one also needs to know the impact of voltage/frequency scaling on thread's performance gain and power usage. Thread frequency scalability prediction can give a good estimate of the performance improvements that DVFS can provide to each thread. We present an active load balancing algorithm which uses thread criticality and frequency scalability prediction to get the maximum possible performance benefit in an energy efficient manner. Results show that balancing the load in an accurate way can give energy savings as high as 10% with minimal performance loss as compared to running all cores at high frequency.

DEDICATION

To my family and friends

ACKNOWLEDGEMENTS

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. Jiang Hu and Dr. Daniel Jimenez for serving on my committee and providing useful feedback on my work.

I would like to take this opportunity to acknowledge all the help I 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 the extensive learning I gained from discussions with my faculty.

NOMENCLATURE

CMP	**C**hip **M**ulti **P**rocessor
MPSOC	**M**ulti **P**rocessor **S**ystem **O**n **C**hip
DVFS	**D**ynamic **V**oltage **F**requency **S**caling
TLP	**T**hread **L**evel **P**arallelism
PCSLB	**P**redicting thread **C**riticality and frequency **S**calability for active **L**oad **B**alancing
FS	**F**requency **S**calability
V	**V**oltage
F	**F**requency
r	Pearson's correlation coefficient
ρ	Spearman's correlation coefficient
SST	**S**um of **S**quares **T**otal
SSE	**S**um of **S**quares **E**rror
RMSE	**R**oot **M**ean **S**quare **E**rror

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Performance growth in past had mostly come from increasing transistor density with the help of process technology scaling. Performance per watt saw almost an exponential increase due to the phenomenon known as Dennard Scaling [7]. Throughout 1990s the microprocessor frequency was the sole metric for performance. In that era of the Megahertz wars, higher frequency meant a faster and better computer. With advancements in process technology, the size of the chip decreased while transistors per chip increased. This together with the continuous frequency increase boosted the power dissipation to threatening levels. This led to a paradigm shift to Megawatts war with power becoming a primary design constraint. As speeding up processor frequency further was no longer an option, computer architects moved to a new approach. Adding multiple processing cores on the same chip would give better throughput while keeping the power under the limits. This search for more performance and limitations of single core processors, gave rise to Chip Multiprocessor (CMP) technology. In recent times, manufacturers have tried to put more and more cores on chip, which gives more performance but on the cost of increased simultaneous active switching, thus again posing power threats. Recently, mobile industry has also shown a lot of interest in CMPs, which puts them into more energy and power constrained environment. This has posed a serious limit to the multi-core scaling, resulting in some part of the chip inactive most of the time, know as Dark Silicon [10]. These limitations have caused researchers to move towards more power and energy efficient designs.

Chip multiprocessors can provide high throughput by utilizing the thread level parallelism (TLP) in the applications. The programs are split into multiple independent tasks which can run on different cores simultaneously. This puts a great onus on the software to provide enough parallelism to efficiently use the available resources on chip. A typical parallel program needs certain constructs such as semaphores, locks (only certain number of threads are allowed to acquire them, others have to wait), barriers(all threads have to reach the barrier before any thread can get past it) and critical sections (executed by only one thread at a time), to maintain the correctness of the program. The thread stall times caused by these synchronization primitives can become the bottleneck and determine the overall speed up achieved by the parallel program, as stated by Amdhal's law [27]. It also causes wastage of energy as thread just spins while waiting for other threads.

1.1 Thread criticality prediction

The imbalance between the threads caused by the synchronization constructs discussed above can cause significant performance and energy degradation. The threads which are responsible for causing this imbalance are termed as critical threads. The aim of thread criticality prediction is to identify the critical threads that cause other threads to wait. These detected critical threads can be potential candidates for various other performance and power optimization techniques. The critical thread can be run on a core running on higher frequency [1], migrated to a more advanced and high performance core [32] in an asymmetric multiprocessor domain or provided with prioritized resource allocation in a shared resource environment [9]. This research project focuses on using per core DVFS to scale voltage and frequency [14] [31]. Critical threads can be run on higher voltage/frequency to get maximum performance benefit or other non critical threads can be slowed down to save energy. In the lifetime of a program, multiple different threads can become critical based on the program behavior, so a dynamic algorithm is more appropriate as it can adapt to the varying program characteristics. Bois et al. in [8] have presented an algorithm which detects the load imbalance by identifying the time spent by a thread in doing useful work and the number of other threads it causes to wait due to synchronization, to determine the criticality of the thread. The core running the critical thread is then operated on the higher voltage/frequency level to achieve better performance. This approach however neglects the impact of the voltage/frequency scaling on the thread's performance and energy consumption, which makes this algorithm energy inefficient and inaccurate in achieving a perfect balance.

1.2 Motivation for frequency scalability prediction

Thread criticality prediction alone can not be efficient in making DVFS decisions as there are phases in the program which do not depend on voltage/frequency scaling of the core. As described by David et al. in [17] and other works[26],[6], the execution of a program can be divided into scalable ($\mathrm{T}_{\text{scalable}}$) and non scalable ($\mathrm{T}_{\text{non\_scalable}}$) phases. Frequency scalability factor can be defined as:

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

Scalable phase of the program can be denoted by the cpu intensive portion which can be sped up in proportion to frequency scaling, whereas the non scalable part is due to the memory intensive behaviour caused by off core accesses and thus independent of the core frequency scaling. To get more insight, we conducted a study of the frequency scalability variations in state of the art multithreaded benchmarks, PARSEC [3] and SPLASH-2 [33], by using the frequency scalability model derived in section 3. Figures 1.1-1.6 show per core frequency scalability variation of ocean_cp, lu_ncb,fmm from SPLASH-2 and bodytrack, dedup, fluidanimate from PARSEC on a 4 core machine. As can be seen, we can clearly distinguish the scalable and non scalable phases of the program. We have also shown the critical thread in each section based on the criticality stacks algorithm [8]. It can be seen that there are many cases where a thread is critical as well as non-scalable, so scaling it to the highest frequency as done in [8], will not be efficient.

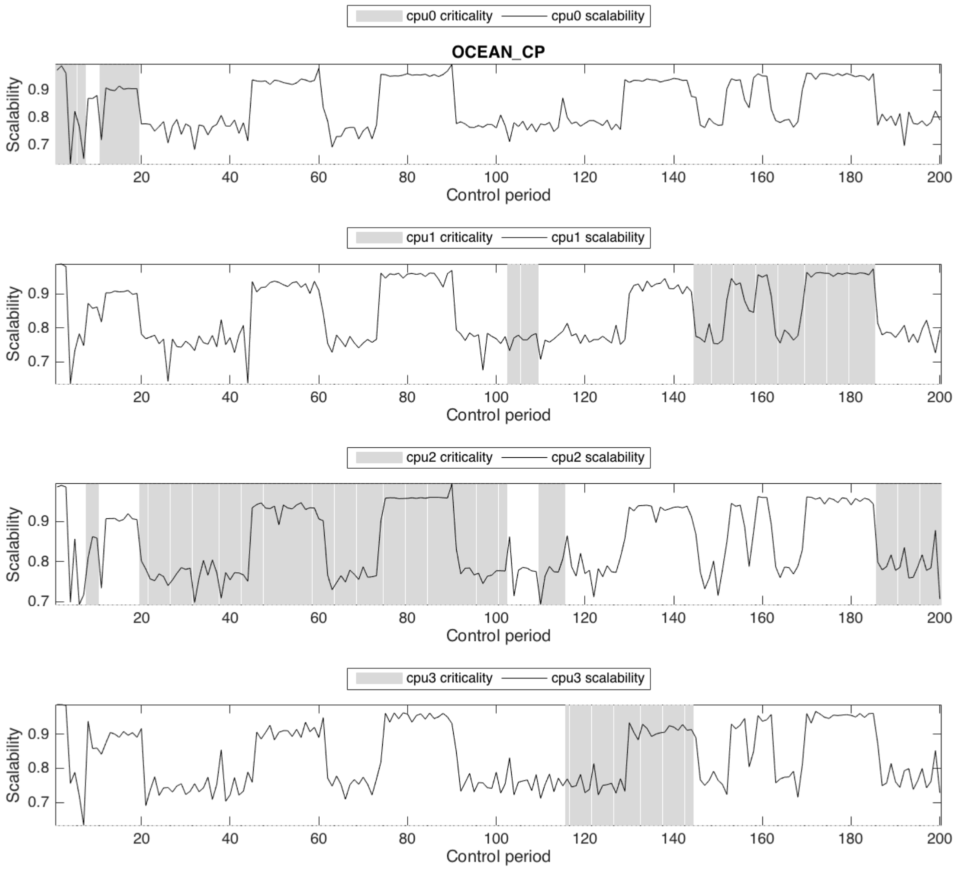

Figure 1.1: Criticality and frequency scalability variation in ocean_cp

In order to check how does the thread criticality prediction reacts to the frequency scalability variation, we ran the criticality stacks algorithm [8] on scalable and

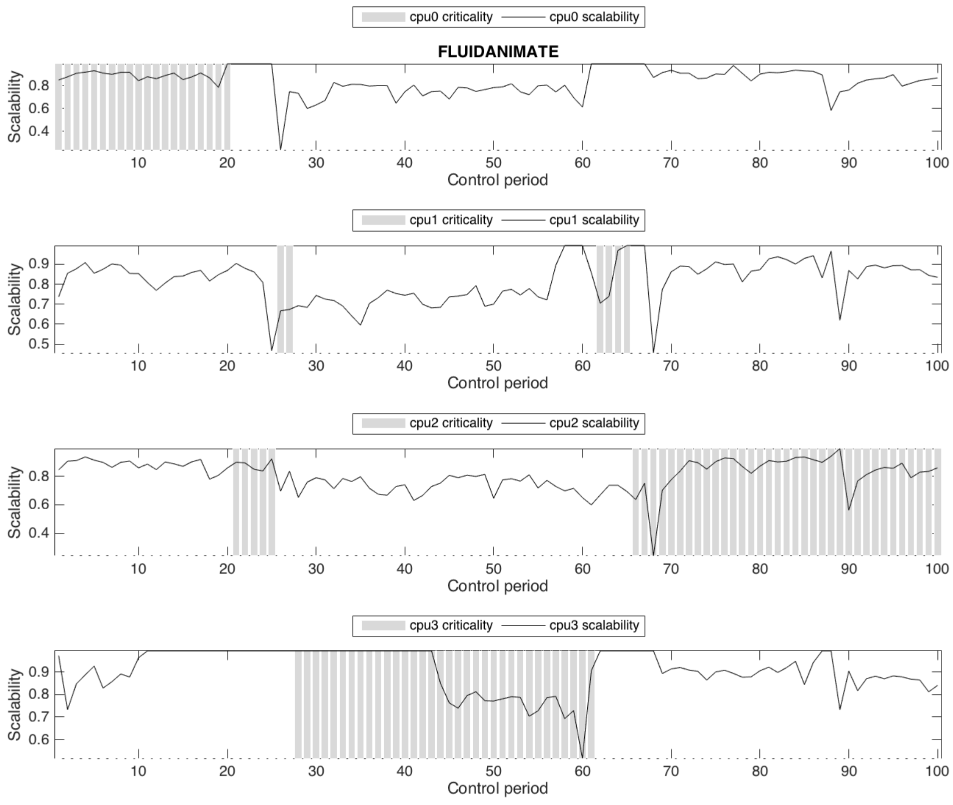

Figure 1.2: Criticality and frequency scalability variation in fluidanimate

non_scalable phase for ocean_cp benchmark to compare their performance, power and energy. Figure 1.7 shows that the performance and energy improvement of the criticality stack drops significantly for non_scalable phase while still consuming almost same power and energy. This shows that thread criticality prediction is not sufficient for making DVFS decisions. To further strengthen the claim, we ran the same experiments but with very low offchip latencies, to neglect the effect of non_scalability. As shown in figure 1.8, the criticality stack performs equally well for scalable and non_scalable phases. This explains that thread criticality prediction neglects the fre-

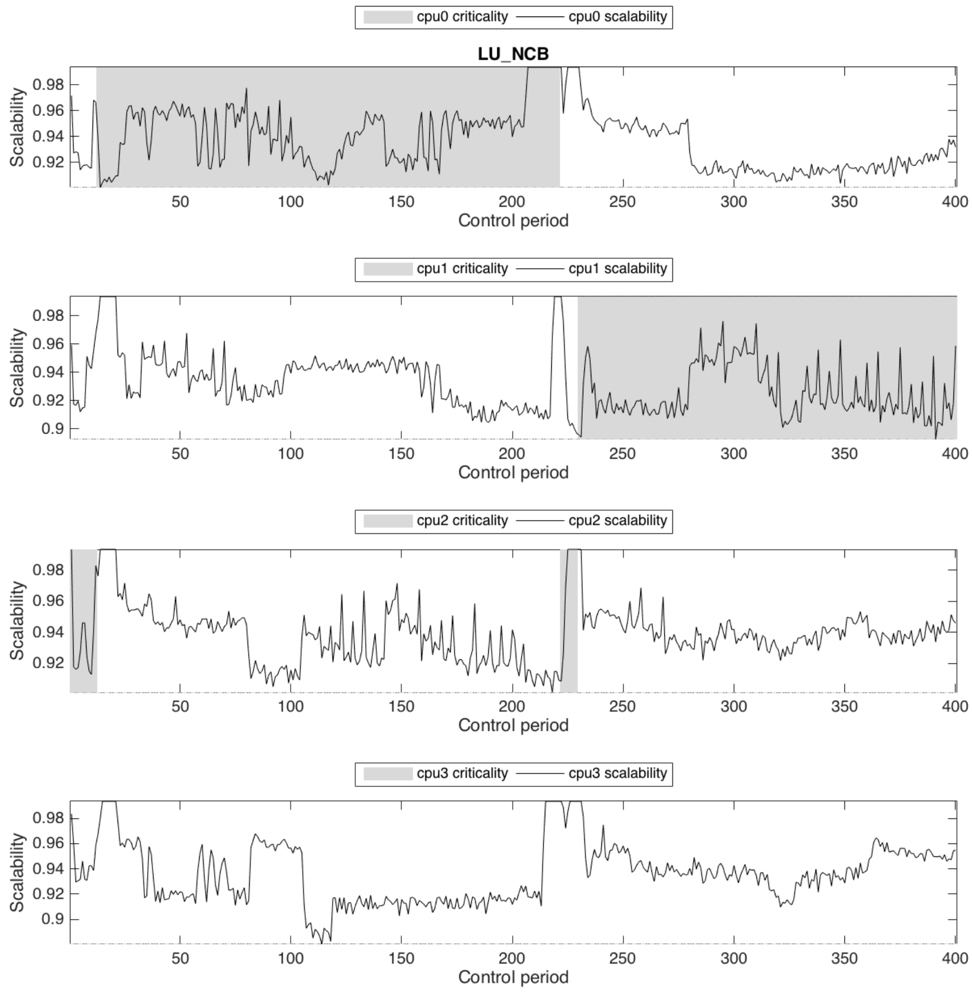

Figure 1.3: Criticality and frequency scalability variation in lu_ncb

quency scalability variation in the program and thus make wrong scaling decisions costing heavily on power and energy without getting much performance benefit. This approach is thus ineffective in attaining a perfect load balance across the threads.

In this work we develop an accurate load balancing algorithm which utilizes thread criticality and frequency scalability prediction. Determining how thread performance

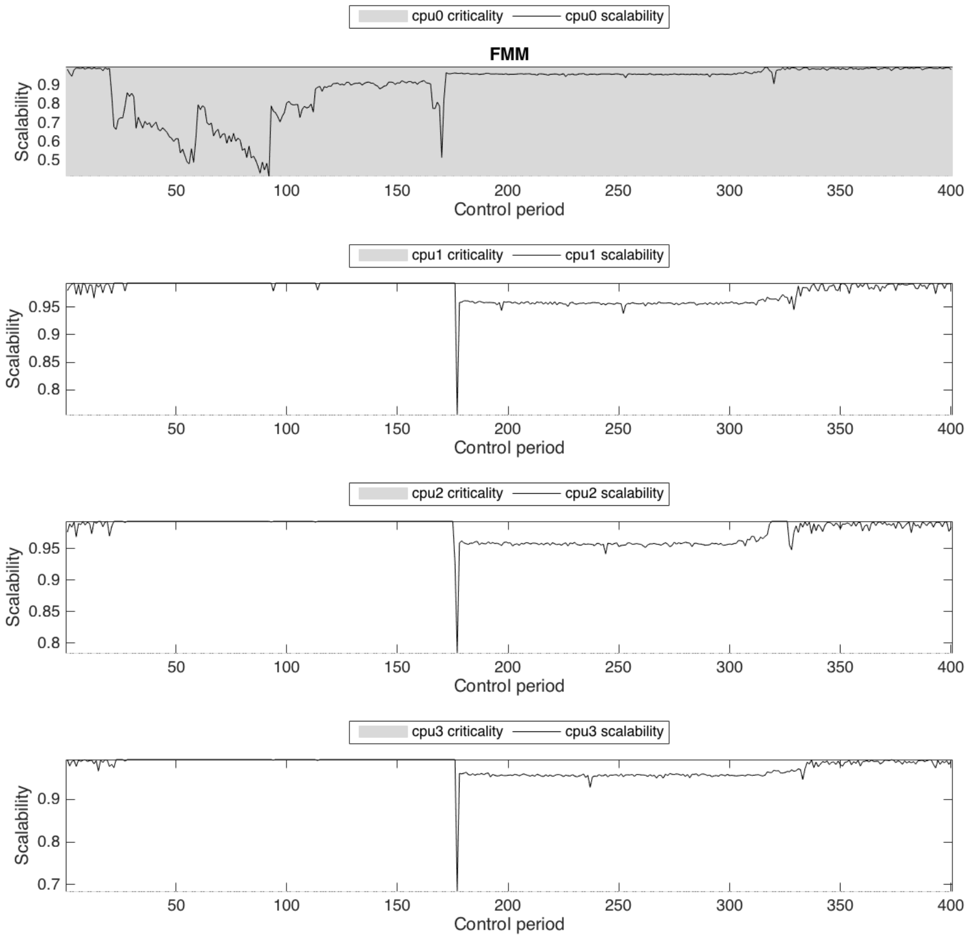

Figure 1.4: Criticality and frequency scalability variation in fmm

scales with different voltage/frequency can give useful insights that can be used to decide an optimal operating point for critical as well as non-critical threads. Such an algorithm can provide maximum performance benefit while still giving energy and power savings. On these lines, following are the key contributions of this work:

- We develop a generalized frequency scalability model for state of the art multithreaded benchmarks using static analysis and curve fitting.

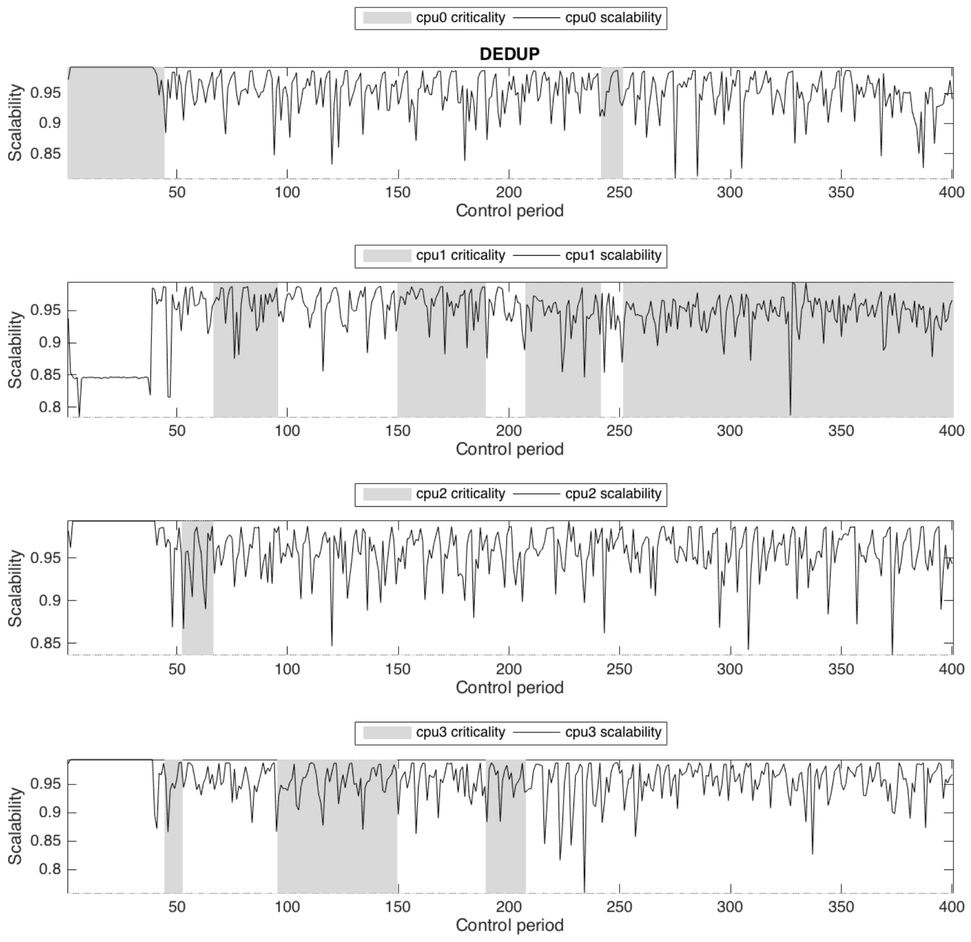

Figure 1.5: Criticality and frequency scalability variation in dedup

- An accurate load balancing algorithm is orchestrated using thread criticality [8] and frequency scalability prediction, as provided by the above model.
- We present the performance, power and energy results and compare them with a famous prior work [15].

The results in section 5 show that having an accurate load balancing algorithm

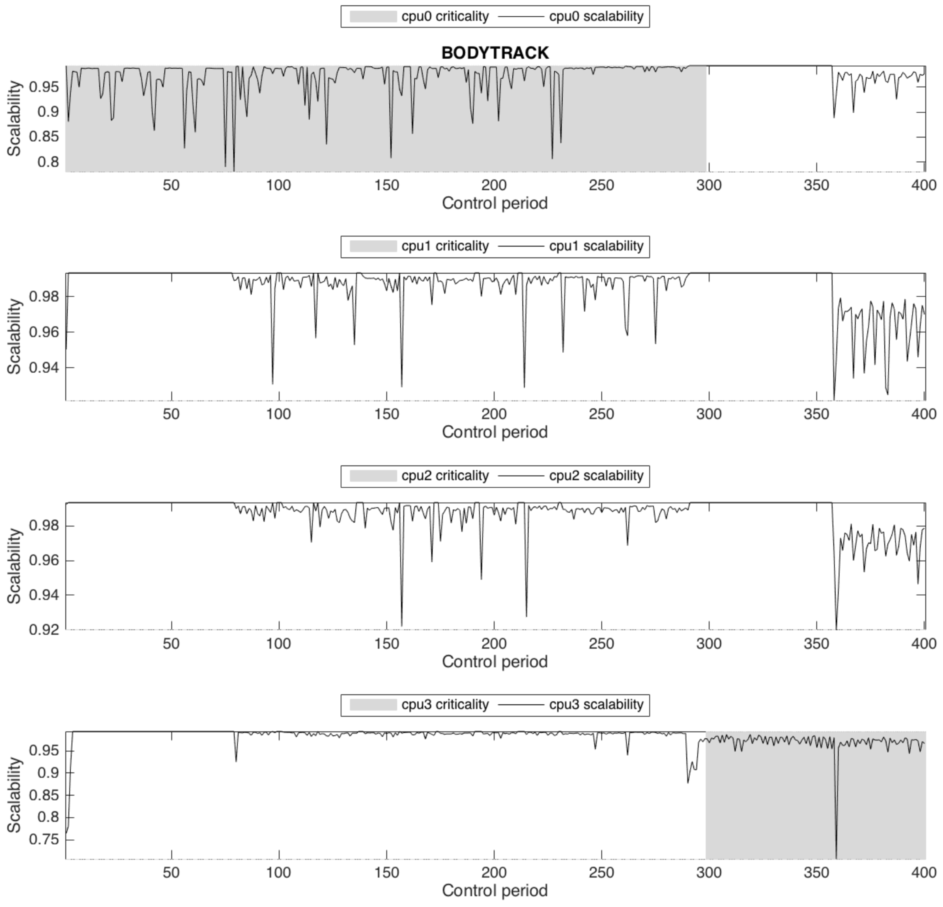

Figure 1.6: Criticality and frequency scalability variation in bodytrack

can give energy savings as high as 10% with minimal performance loss as compared to the upper bound, where all cores run at the highest frequency available.

The rest of the thesis is organized as follows. Section 2 gives a literature survey of prior work on DVFS, thread criticality and scalability analysis. Section 3 discusses about the static scalability modeling to accurately predict performance of parallel workloads. Section 4 provides description of our active load balancing algorithm (PCSLB), using thread criticality prediction and scalability model developed

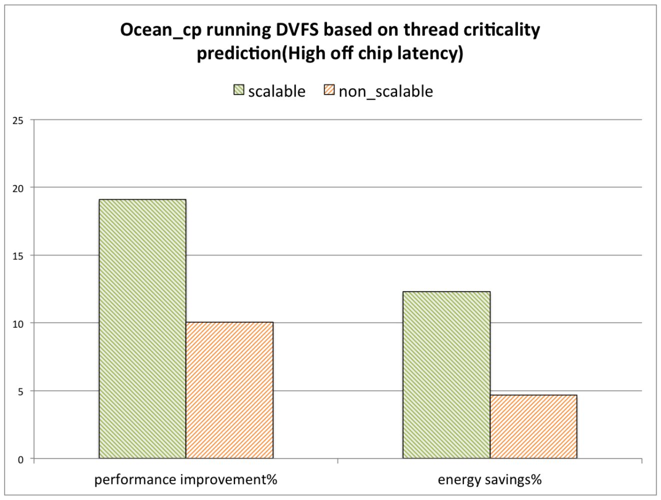

Figure 1.7: Performance and energy comparison for scalable and non_scalable phase in ocean_cp

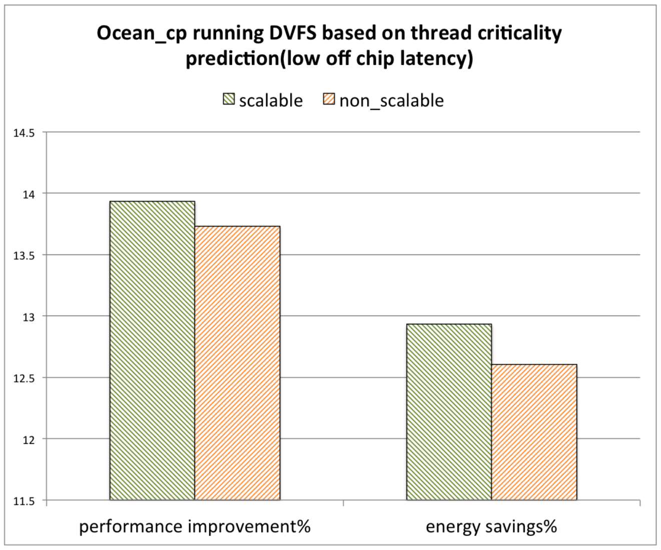

Figure 1.8: Performance and energy comparison for scalable and non_scalable phase in ocean_cp with low off chip latencies

in section 3. In section 5, we present the experimental setup and results showing the performance, power and energy analysis of PCSLB and comparison with prior work [15]. At the end we present the Conclusion and future work in section 6.

2. PREVIOUS WORK

Performance and power improvements in multicore architectures have been a widely studied topic. There are many ways researchers have tried to use Dynamic Voltage and Frequency Scaling to get better performance and save energy. Thread criticality prediction is one such approach for parallel workloads. This section will go over some historical works on DVFS followed by the research in the area of thread criticality prediction and frequency scalability analysis.

2.1 Dynamic voltage frequency scaling on chip multiprocessors

During the ”Megahertz war” around late 90’s, the microprocessor clock frequency was continuously rising. Intel introduced Pentium 4 in 2000 which could run on clock frequency as high as 2.8 GHz. This frequency rise along with increasing transistors on chip, adversely affected the clock distribution network complexity. To get around this problem and continue increasing the clock frequency further, researchers started migrating from single clocked globally synchronous systems to globally-asynchronous, locally-synchronous (GALS) systems [16]. One approach that uses GALS clocking style is Multiple Clock Domain (MCD) processors [30] , where each functional block is clocked separately. This technique not only solves the problem of clock distribution but also introduces a new degree of freedom: each domain can work on independent voltage and frequency levels. This was how Dynamic Voltage and Frequency Scaling became an important approach for power/performance improvement. Since then the researchers have been trying to solve the open problem of ”when and how to change the voltage and frequency” to maximize performance in a given power budget.

Initially, offline DVFS schemes were devised to find the right switching times. Semeraro et al. in [30] uses reconfiguration tools to construct a dependence directed acyclic graph (DAG) from the program trace which is then analyzed to find the critical path and the paths with "slack". Accordingly frequency and voltage are scaled to save energy and maximize the performance. Magklis et al. [25] use profiling to insert reconfiguration instructions into the program which decide the voltage/frequency levels while the program is running. This required the application to go through "training runs" to identify phases of different voltage/frequency settings.

Even though the offline techniques save on the hardware complexities, they are very application specific. In contrast, online DVFS techniques captures runtime characteristics which makes them more realistic and compatible with all applications. Semeraro et al. [29] proposed such an online-algorithm to find the optimal domain frequency by using processor queue utilization as a metric. The algorithm, which they call as "Attack/Decay", exploits the co-relation between the number of entries in the queues with the optimal frequency for that domain. The idea behind this technique is that by looking at the behavior of the issue queues, we can decide what frequency that domain should run on. For example if the number of entries in a queue is stationary, that means the receiver and sender domains are at optimal frequencies. If the queue is emptying that means the sender is slower than the receiver domain and vice versa. Wu et al. [34] asserts that the above technique is heuristic-based as it is based on manually selected rules and threshold values which makes it hard to scale and improve according to the dynamic behavior. They propose a more rigorous and analytical approach to model the queues and the domains. They then formulate linear equations connecting the demand and frequency of a domain and solve them using a Proportion-Integral-Derivative (PID) controller.

Grochowski et al. [13] adds a new direction by looking at the scalar and parallel phases of the code and design a microprocessor that can adapt to them. The idea is that we can decide the amount of energy resources allocated based on the parallelism of the code. The relationship is formalized as:

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

Here EPI is the average energy spent per instruction, IPS is average number of instructions executed per second and P is the power budget which is fixed. For a scalar code, IPS is low so we can Increase the EPI to improve performance without hurting the power. Similarly parallel code has high IPS, so its better to spend less EPI to be in the power limits. This paper estimates the combination of asymmetric cores and DVFS to be most promising design approach to achieve good latency and throughput performance together. So in phases of low parallelism, its better to run the program on the large core with high voltage and frequency, while for parallel code run small cores with low voltage and frequency settings.

Isci et al. [15] introduced the concept of global power manager on chip, that checks the power and performance of each core by using a feedback-control technique. Several different global dynamic power management policies are evaluated and claimed that they perform better than static policies. It also includes per-core DVFS to assign power modes to individual cores. Their best performing policy, MaxBIPS, finds the optimal combination of DVFS levels for all the cores on chip using an exhaustive search and predicts the power and BIPS (Billion Instructions Per Second) values for all the possible combinations of DVFS levels. This can help in extracting the best possible application performance while in the limits of power budget. While this approach works well for few cores, this is not a scalable solution if the number of cores increase. We compare against this work and show that our approach does better in energy savings with less performance cost, by utilizing the information about thread criticality and frequency scalability.

2.2 Thread criticality prediction

Thread criticality is one concept which has been an open problem for researchers since late 2000s. With many parallel applications coming up, workload imbalance has become an important source of energy inefficiencies for CMPs. The threads of a parallel program generally share information among each other which causes these threads to execute at different speeds. So predicting which threads or processes are critical and what are the speed difference from non-critical threads, can offer potential performance increase while maintaining the power of the chip. The very first work in this area was done by Li et al [20] which used the knowledge of the time taken by threads to reach the thrifty barriers in parallel program to calculate per-core frequency. These barriers are used to synchronize the critical sections in parallel programs. The thread reaching the barrier at the last is considered to be critical and the cores running other non-critical threads can be slowed down or shut down to save the chip power. Li at al. [20] shut down the cores reaching the barrier earlier than the critical thread, whereas Liu et al. [23] slow down the non-critical cores using DVFS. While these technique provide considerable power savings, the thread criticality prediction is done by a simple last-value predictor which uses the current barrier stall times, that is the time a core is waiting on the barrier for other cores to arrive, to estimate the future times. So all threads have to run on the normal frequency until the barrier is reached. To resolve this, authors in [5] proposes a method to dynamically check the running threads for the imbalance at intermediate check points which they call as meeting points. This approach was tested with parallel loops and meeting points were inserted at the end of the loop by the software. This helped to count the number of loop iterations completed by a given thread which can be used to calculate the criticality before the barrier is reached.

Significant energy power savings were achieved, but this scheme was only intended for parallel loops. The concept of thread criticality prediction was generalized by Bhattacharjee et al [2]. This paper presents a study of different possible architectural parameters which can have a high impact on thread criticality. Cache miss was found to be the most effective metric since more the cache misses, slower the thread will be. The design consisted of criticality counters which kept a track of the L1 and L2 cache misses for each thread. On each interval, these counter values are scanned to find the critical thread. Even though this work provided a good insight into what can be possible parameters to consider, the dynamic approach using cache misses is not fully convincing as cache misses are mostly invariant to the core frequency. Bois et al [8] provided a different approach for thread criticality prediction with the introduction of criticality stacks. They provided an offline approach which combines active running time of the thread with the number of dependent threads waiting on it. Though this gives a good estimate of the load imbalance among the threads, using it online with DVFS will not be energy efficient as this approach does not take into account the off chip accesses of the threads which makes it unscalable and less sensitive to voltage/frequency. In our approach, we use the load imbalance detected by the criticality stacks and utilize the scalability information of each thread to make accurate DVFS decisions which leads to better load balance.

2.3 Thread frequency scalability prediction

Having a good estimate of how much a thread's performance changes with scaling it's voltage/frequency is very critical in order to make energy and power efficient DVFS decisions. Failure to do that might lead to energy and power wastage and even performance degradation as it can adversely affect the balance between the thread. There have been quite a few works on predicting performance impact of frequency scaling. Lee et al. [19] did an extensive study of various microarchitectural parameters which can accurately predict the performance and power for different applications. They derived regression models to validate the strengths and significance of different predictors. In contrast to an offline analysis, Kihwan Choi et. al. [6] proposed a dynamic regression based algorithm which models the frequency scalability of a chip by the ratio of off chip access time to on chip time spent on computation. Another very intuitive way to predict the frequency scalability which was presented in [11] was to look at the commit bandwidth. The stalls due to unable to commit instructions can be one of the cause of the non scalability. These stalls can be due to outstanding memory instructions which are non scalable. Commit stalls are not an accurate prediction technique as there might be independent instructions committing under an outstanding memory access. Leading loads [12] [18] [28], provides an abstract view of non-scalability by taking into account the off chip latency of the first load miss in the series of load instructions. The latencies of such leading loads are accumulated to predict the non-scaling time. Rustam et. al. [26] asserts that the leading load assumes a constant memory access times which can be inaccurate for the realistic memories. They proposed an algorithm CRIT, which accumulates the variable latency of load misses that are on the critical path of dependent memory instructions. All these techniques have been mostly studied for single threaded applications. Multi-threaded applications behave very differently due to the inherent imbalance caused by the inter thread synchronizations. In a very recent work, Akram et. al. extended CRIT algorithm[26] for multi threaded workload by taking into account the synchronizations between the threads, to predict the system performance on different frequencies.

In our work, we developed a regression based frequency scalability model similar to [19] for multi-threaded workloads and used it to predict each thread's performance on different frequency settings. We will see in the next section how the model differ for parallel applications in contrast to the model developed for single threaded workloads [19].

3. FREQUENCY SCALABILITY MODELING

This section presents our study of various micro architectural parameters that can affect the frequency scalability of a thread in widely used parallel workloads [3] [33], similar to what was done by Lee et al. in [19] for single threaded applications.

Frequency scalability tells an estimate of how much the performance can be scaled directly with frequency. Given two frequencies F_1 and F_2 such that $F_1 > F_2$, and time taken by the application corresponding to these frequencies as T_1 and T_2 respectively, the frequency scalability can be expressed as:

$$FS = \frac{T_2 - T_1}{T_1 * (\frac{F_1}{F_2} - 1)} \quad \mathrm{F}_1 > \mathrm{F}_2 \tag{3.1}$$

Our aim is to predict frequency scalability of a thread dynamically using available hardware monitors. In order to that, we need a linear relationship between scalability and the monitors as follows:

$$FS = \alpha_1 + \alpha_2 p_1 + \alpha_3 p_2 + ... \tag{3.2}$$

In above equation, $[p_1, p_2.....]$ are different predictor variables which are provided by hardware monitors, $[\alpha_2, \alpha_3.....]$ are the coefficients which tell the impact of each predictor on the scalability and α_1 gives the weight of non-modeled part of the program. We will discuss how to model this linear relationship in this section. Hardware specifications we chose for modeling are shown in table 3.2. We selected the predictor variables as shown in table 3.1. Benchmarks from parallel workload suite SPLASH [33] and PARSEC [3] as shown in table 3.3 were selected for modeling.

Predictor	Description
branch miss rate	Number of branches mispredicted per total branches
commit per cycle	Number of instructions committed per cycle
commit bw stalls	Stalls due to unable to commit an instruction
conflictingLoads	Dependent loads unable to issue
conflictingStores	Dependent stores unable to issue
cpu mem ratio	Ratio of cpu instruction to memory instructions
fu busy cnt	Number of times functional unit was busy causing stall
IQFullEvents	Number of times Instruction Queue is full
LQFullEvents	Number of times Load Queue is full
SQFullEvents	Number of times Store Queue is full
load store ratio	Ratio of loads to stores ratio
l1 data MPKI	Total private data cache misses per kilo instruction
l1 inst MPKI	Total private instruction cache misses per kilo instruction
l2 data MPKI	Total shared L2 data misses per kilo instruction
l2 inst MPKI	Total shared L2 instruction misses per kilo instruction
tlb MPKI	Total instruction and data tlb misses per kilo instruction

Table 3.1: Characteristics examined to model frequency scalability

3.1 Sampling

Parallel programs can have varied dynamic behaviour as we showed in section 1.2. The sampling should be done in such a way that can capture information for all these different phases. To find an optimal sampling period, we analyzed different benchmarks and found out that 1 million dynamic instructions are enough to capture any of the different phases. Hence we divided the execution of all benchmarks in sections of 1 million instructions and ran them on a set of frequency range as mentioned in table 3.2. We compared the execution times of corresponding sections

ISA	ARM version 7
no. of cores	4
core type	4-wide out-of-order
frequency range	1.2-3.9 GHz in steps of 0.3Ghz
L1 D-cache	64 KB, private, 2 cycles
L1 I-cache	32 KB, private, 1 cycles
L2 (last level)	2MB, shared, 12 cycles
memory access	100 ns

Table 3.2: Hardware specifications for frequency scalability modeling

and substituted them in equation (3.1) to get the observed performance benefit and recorded their predictor values.

3.2 Model derivation

The most important step in model construction consists of finding significant relationships between the predictor variables and the response by carefully analyzing the data space. Correlation analysis helps in identifying the predictor variables which are highly significant in predicting the response. We then choose the variables having high statistical relationship with the response, to participate in the final step of model evaluation.

3.2.1 Pearson's correlation coefficient

Pearson's correlation gives a linear relationship between the predictor variable and the response. The formulation for n sample values is given by:

$$r = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2 \Sigma(y_i - \bar{y})^2}} \tag{3.3}$$

where,

Suite	Benchmark	Input
PARSEC	Bodytrack	simmedium
	Dedup	simmedium
	Ferret	simmedium
	Streamcluster	simmedium
	x264	simmedium
SPLASH-2	barnes	simmedium
	cholesky	simmedium
	fmm	simmedium
	lu_ncb	simmedium
	ocean_cp	simmedium

Table 3.3: Benchmark specifications

r is pearson's coefficient

Range: $-1 \leq r \leq 1$

x and y is the dataset of two variables having n samples

$\bar{x}$ and $\bar{y}$ are the sample mean given by $\frac{1}{n}\Sigma(x_i)$ and $\frac{1}{n}\Sigma(y_i)$ respectively

However, pearson's correlation assumes that the distribution of x and y are known and will give good results only when the changes in x are consistent with changes in y. In order to find correlation between two variables such that the two variables have a monotonic relationship where they change in a similar trend without a need of a constant rate, Spearman's rank order correlation is more suitable.

3.2.2 Spearman rank correlation coefficient

Spearman's coefficient ρ quantifies monotonic relationship between X and Y such that when Y increases, either X always increase, decrease or remain unchanged. As it is based on ranked values instead of the actual values, it is independent of the distribution of the variables. To calculate ρ, the absolute values of x and y are converted to ranked columns X and Y and difference between them is calculated and stored in d. Assuming n ranks are distinct integers, ρ can be computed as follows:

$$\rho = 1 - \frac{6 * \Sigma(d_i)^2}{n * (n^2 - 1)} \tag{3.4}$$

We use squared Spearman's coefficient to quantify the predictor strengths as the distribution of the variables are unknown. Figure 3.1 and 3.2 shows each predictor's strength for PARSEC [3] and SPLASH [33] benchmark respectively. Finally figure 3.3 shows the predictor strengths when the samples from all the benchmarks are combined.

There are some important observations to be made from these graphs. L1 data MPKI, l2 data MPKI, tlb MPKI and LQFullEvents have high significance in most benchmarks. Number of committed instructions per cycle show good correlation in some benchmarks. Instruction L1 and L2 MPKI are less deterministic of the performance as they are very rare. Other parameters are not consistent so we neglect them. Looking at the consolidated correlation of all the benchmarks together, we choose l1 data MPKI, l2 data MPKI, LQFullEvents and tlb MPKI as potential candidates for the frequency scalability equation. In contrast to the model developed by Lee et al. [19], we found that committed instructions is not a significant predictor for parallel workloads as there can be dynamic instructions due to spinning while a thread is trying to acquire a lock. These instructions don't contribute towards the performance.

3.3 Model assessment

We got four variables from our analysis in previous section which are potential candidates for the frequency scalability equation. In this section we will model frequency scalability with different combinations of these variables and evaluate them

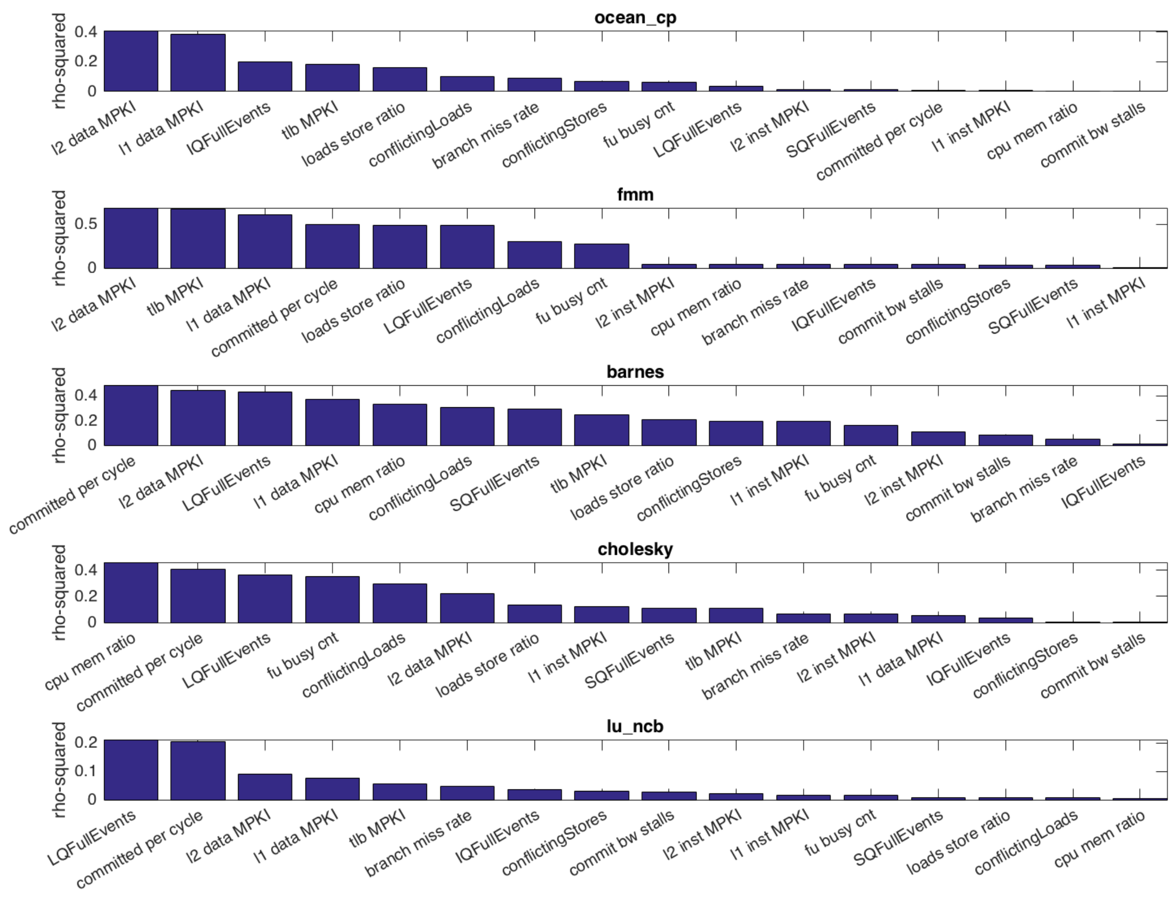

Figure 3.1: Predictor strength for SPLASH benchmarks

by using multiple correlation statistic (R^2) and Root Mean Square Error (RMSE). These statistics are based on two sum of squares as follows:

- Sum of Squares Total (SST) estimates the variance between data and mean.
  - $\text{SST} = \sum_{i=1}^{n}(y_i - \frac{1}{n}\sum_{i=1}^{n} y_i)$
- Sum of Squares Error (SSE) measures the error between the actual data and the predicted value.

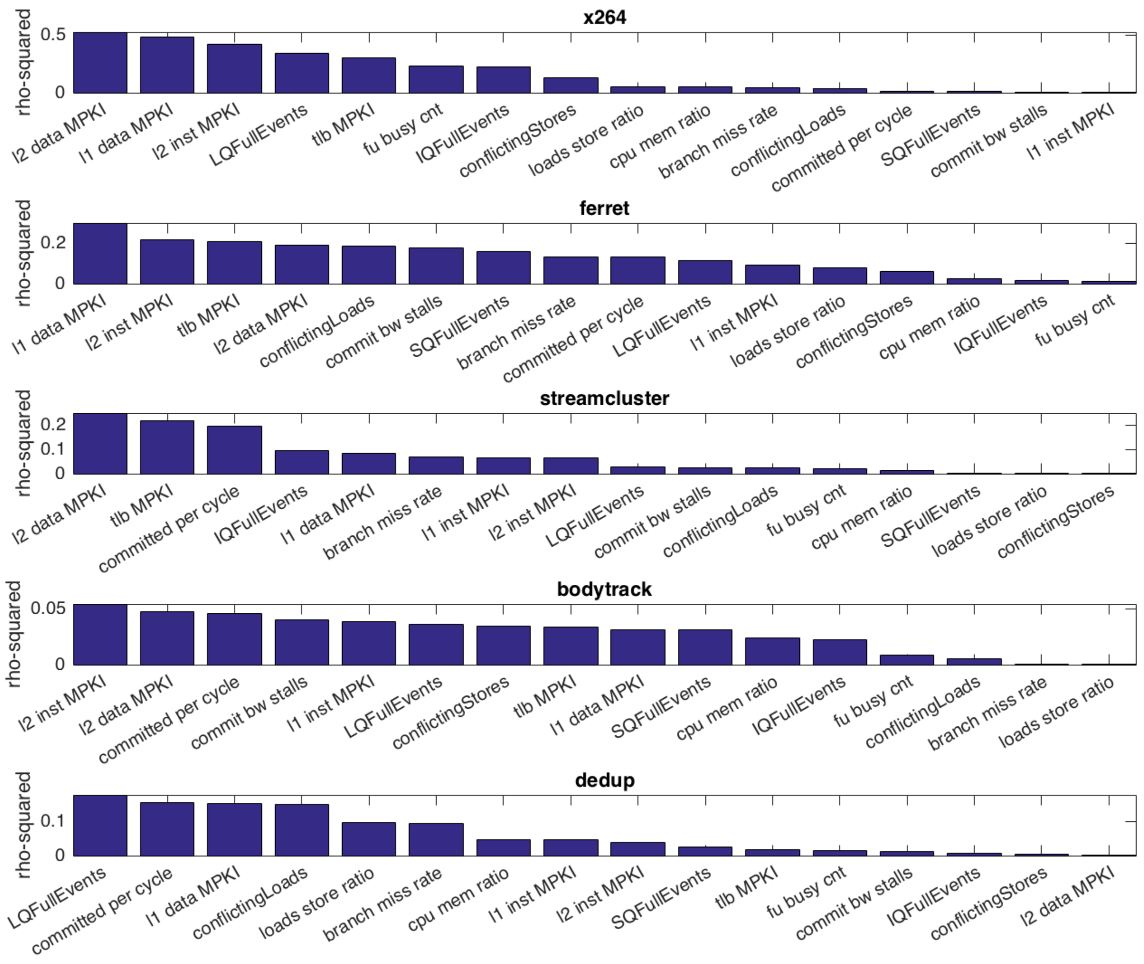

Figure 3.2: Predictor strength for PARSEC benchmarks

– SSE $= \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

3.3.1 R-squared and adjusted r-squared

The improvement of the prediction of the regression model from the mean model is given by the difference of SST and SSE. R^2 is obtained from dividing this difference

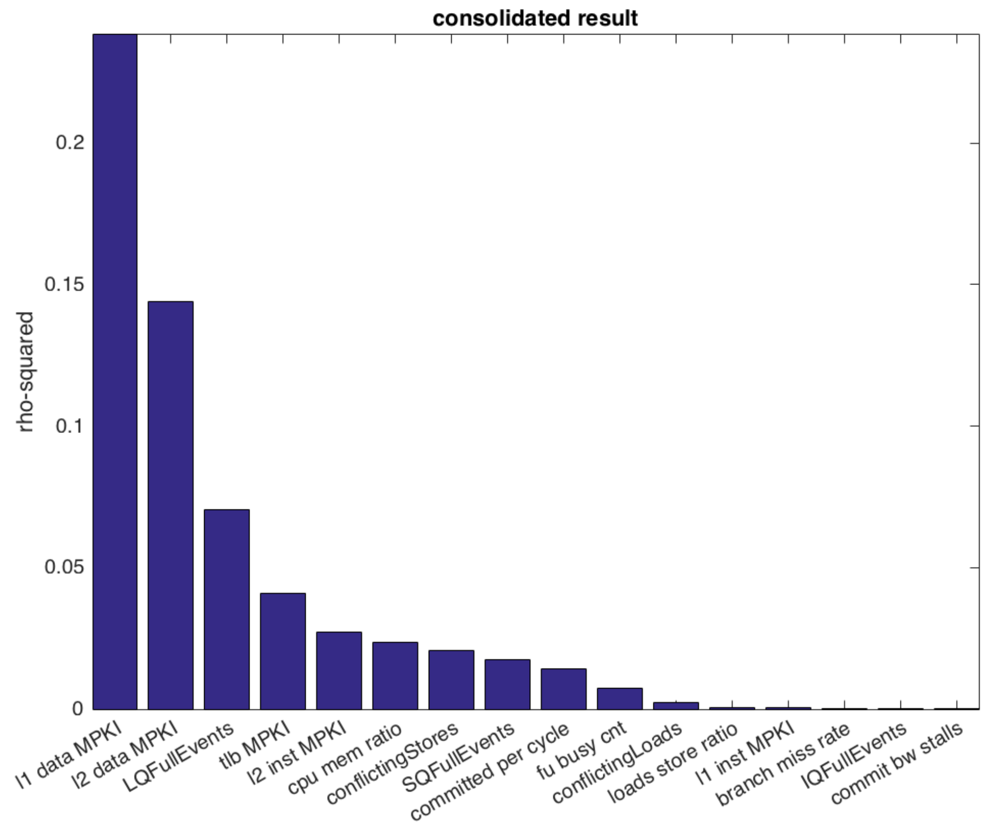

Figure 3.3: Predictor strength for all benchmarks combined

from SST as follows:

$$R^2 = 1 - \frac{SSE}{SST} \tag{3.5}$$

R-squared estimates the goodness of the fit on the scale of 0 to 1 with high values suggesting a good fit. One drawback of R-squared is that it increases with increase in the number of predictor variables. So a value every close to 1 suggests over fitting. To solve this, we consider Adjusted R-squared which also models the degree of freedom. So an increase in the number of predictors will decrease the Adjusted R-squared value.

3.3.2 Root mean square error

RMSE measures the accuracy of the prediction. As we are developing a model for scalability prediction, we need to know how close the predicted scalability is to the observed value. RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{3.6}$$

To evaluate a model we try to minimize the RMSE as much as possible. So in this way, RMSE gives an absolute measure of the fitness of the model.

We evaluate the frequency scalability models from various possible combinations of the set of four variables {l1 data MPKI,l2 data MPKI, LQFullEvents, tlb MPKI} and plot their Adjusted R-squared and RMSE values in fig 3.4. As can be seen, we get the best model with variables {l1 data MPKI,l2 data MPKI} with the highest R^2 and minimum RMSE.

The final scalability equation is as follows:

$$FS = \alpha_1 + \alpha_2 l1dataMPKI + \alpha_3 l2dataMPKI \tag{3.7}$$

with their coefficients as in table 3.4. α_2 and α_3 are negative as the scalability will decrease if either of the l1 or l2 data misses increase. α_1 is positive, indicating the scalability value when there are no l1 and l2 data misses.

	Estimate	SE	tStat	pValue
α_1	0.99409	4.3284e-05	22967	0
α_2	-0.0054498	1.3951e-05	-390.63	0
α_3	-0.089083	7.3362e-05	-1214.3	0

Table 3.4: Estimated coefficients for equation (3.7)

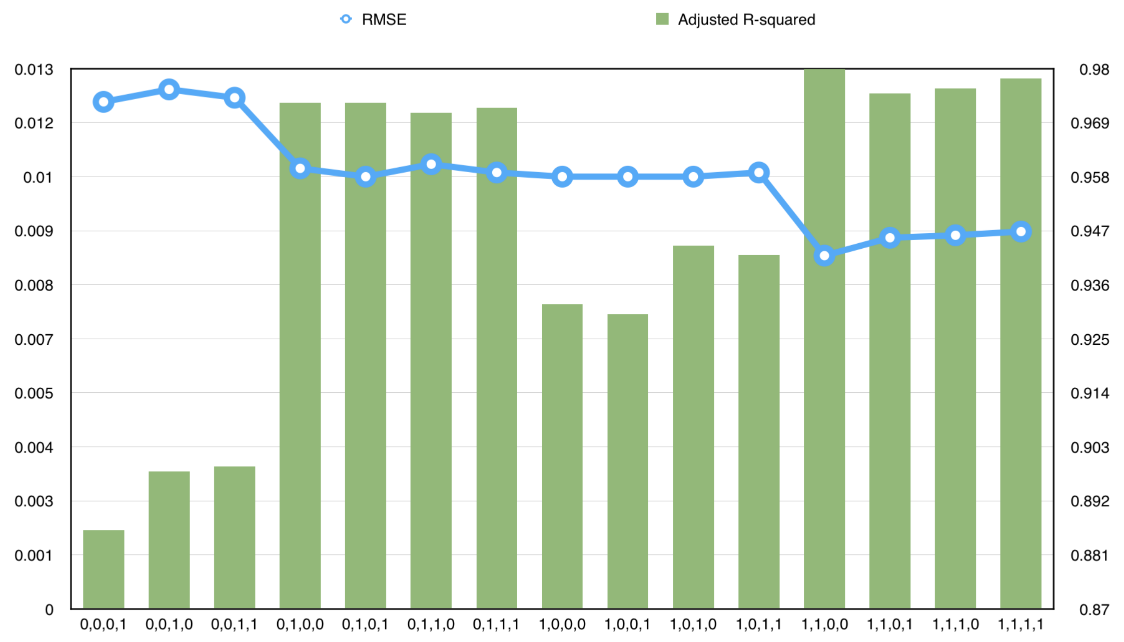

Figure 3.4: Adjusted R-squared and RMSE for the set of variables {l1 data MPKI,l2 data MPKI, LQFullEvents, tlb MPKI} with 1 indicating the presence and 0 indicating the absence of the variable

4. ACTIVE LOAD BALANCING ALGORITHM

In this section we explain our load balancing algorithm which uses thread criticality prediction algorithm from criticality stacks [8] to detect the load imbalance between threads and frequency scalability prediction to predict the performance of a thread on different frequency.

4.1 Thread criticality algorithm

In this section we explain how we detect and measure the load imbalance in a parallel program, using criticality stacks [8]. The whole execution time of the program is divided into different time intervals. A thread will be in active state if it is doing useful work and becomes inactive if it is spinning while trying to acquire a lock. Every time any thread changes its state, a new interval begins. Each interval's execution time is divided by the number of active threads in that interval and allotted to each active thread's criticality count. In this way, criticality stacks takes into account, the information of a thread's useful work and number of threads waiting on it.

Figure 4.1 shows an example of load imbalance caused in 4 threads. The criticality count at the end of the control period shows that thread 3 is critical with the highest criticality count followed by thread 2, 1 and 0. To quantify the load imbalance, we define "Slack" as the measure of how much a non-critical thread is ahead of the critical thread. Slack can be calculated in percentage as follows:-

$$Slack_i\% = \frac{C_{critical} - C_i}{C_{critical}} * 100 \tag{4.1}$$

where, i is thread number and C is Criticality count

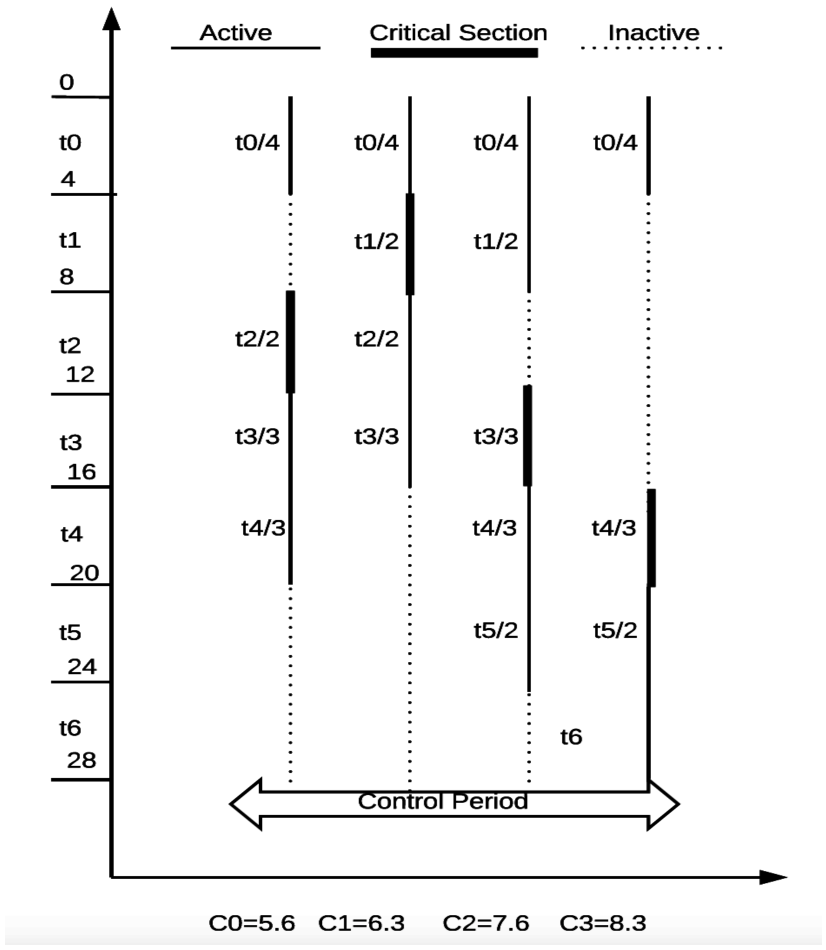

Figure 4.1: Criticality stack example

For the example in figure 4.1, slack for thread 0 is 32.5%, for thread 1 it is 24% and thread 2 has a slack of 8.4%. Looking at each thread's slack percentages, we can tell how much imbalance the program has, at any point of time. This information can help us making accurate DVFS decisions as we will discuss later.

4.2 Thread performance prediction

In order to make accurate DVFS decisions for each core, apart from the load imbalance information, we also need to know about the impact of frequency scaling on each thread's performance. In this section, we will discuss how we predict the performance using the frequency scalability modeling discussed in section 3.

Section 3 models the frequency scalability equation (3.7) which predicts the scalability of the thread based on the L1 and L2 (last level) data MPKI. We put the estimated scalability value in the below equation derived from equation (3.1) to predict the execution time on the target frequency:

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

where, $T_{current}$, $F_{current}$ are the current interval's execution time and frequency of the thread and T_{target} is the estimated execution time of the thread when run on F_{target} frequency.

To quantify the performance benefit we get from scaling the frequency of a thread, we define "ExpectedGain" as percentage change in the performance in terms of execution time. The formulation for ExpectedGain is as follows:

$$ExpectedGain\% = \frac{T_{current} - T_{target}}{T_{current}} * 100 \tag{4.3}$$

where, $T_{current}$ and T_{target} are the execution times of the thread at the current and target frequency respectively.

ExpectedGain will be positive if $T_{target} < T_{current}$ and will be negative if $T_{target} > T_{current}$. A negative ExpectedGain means that $F_{target} < F_{current}$ and thus the performance dropped.

4.3 Achieving the balance

In above sections, we have introduced ways to quantify the load imbalance and the performance gain by scaling the frequency. We will now put them together to develop a dynamic algorithm which will actively try to balance the load with high accuracy. Achieving the balance is vital for power and energy savings while getting guaranteed performance benefit, as we will see from the results in the next section.

Algorithm 1 gives a simple and intuitive way to solve the load imbalance problem by intelligently deciding the operating frequency of each thread by taking into account it's Slack with respect to the critical thread and ExpectedGain at different frequencies.

We run algorithm 1 at the end of every control period to decide frequency of each core for the next control period. We set the control period to 3ms as it is close to OS scheduling period and thus the DVFS can be handled by OS. The process followed by our algorithm can be described as:

1. At the start of every control period, we record the criticality count for each thread and calculate their Slack percentages as discussed in section 4.1.

2. We also calculate each thread's frequency scalability factor according to equation (3.7). In order to filter out sudden changes in the scalability, we calculate exponential moving average of the scalability as follows:

   - $FrequencyScalability_i = \alpha * FrequencyScalability_{i_{current}} + (1 - \alpha) * FrequencyScalability_{i-1}$

   - Here $FrequencyScalability_{i_{current}}$ is the frequency scalability calculated in current interval for thread i and $FrequencyScalability_{i-1}$ is the exponential average calculated in the previous interval. α represents the

Algorithm 1 Active 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:     $FrequencyScalability_{i_{current}} \leftarrow$ Calculate scalability for each thread using equation (3.7)
 4:     $FrequencyScalability_i \leftarrow \frac{FrequencyScalability_{i_{current}} + FrequencyScalability_{i-1}}{2}$    ▷ exponential moving average
 5: end for
 6: $T_{F_{max}}(critical) \leftarrow$ Predicted execution time for critical thread at $F_{max}$ using equation (4.2)
 7: $ExpectedGain_{critical} \leftarrow$ Predicted ExpectedGain for critical thread when run at $F_{max}$ using equation (4.3)
 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 (4.3)
11:     $F_{target_i} \leftarrow$ Calculate Target frequency using $T_{target_i}$ from equation (4.2)
12: end for
```

smoothing factor, which we fix at 0.5.

3. Frequency for the critical thread is set to the highest frequency (F_{max}) and it's ExpectedGain is predicted ($ExpectedGain_{critical}$).

4. ExpectedGain required for each thread ($ExpectedGain_i$) is calculated by subtracting thread's slack ($Slack_i$) from ExpectedGain of the critical thread

($\text{ExpectedGain}_{\text{critical}}$)

5. Once we have the required ExpectedGain for each thread, we calculate the T_{target} from equation (4.3) which is substituted in equation (4.2) along with T_{current}, F_{current} and FrequencyScalability, to get the target frequency (F_{target}) for each thread.

6. We change the frequency of all the threads to the target frequencies calculated above.

We run the critical thread always at the F_{max} to maximize the performance, as our aim is to get minimum performance loss compared to running all threads at F_{max}, with maximizing the power and energy savings. We then decide optimal frequency for each non-critical thread such that the load is always balanced. This decision is based on the difference between the ExpectedGain achieved by the critical thread when run at the highest frequency and the thread's own Slack (Slack_{i}), as shown in step 4. This difference can be positive as well as negative. A positive difference would mean that scaling critical thread to F_{max} will give more performance benefit to the critical thread than the slack for the thread i. If we don't raise the frequency of thread i, we will again create an imbalance. So we raise the frequency of thread i such that it's ExpectedGain equals the difference. On a contrast, a negative difference would mean that even if we scale the critical thread to F_{max}, the load imbalance will still be there. So it is wise to scale down the frequency of thread i such that the performance degradation equals the difference. This will save power/energy without hurting the performance.

5. EXPERIMENTAL RESULTS

In this section we will discuss the experimental set up, followed by the results of our load balancing algorithm and comparison with prior works.

5.1 Experimental setup

The baseline hardware is a 4 core CMP, ARM ISA machine with 2 level cache hierarchy as specified in table 5.1. Each core's private cache is split into Icache (32KB) and Dcache(62KB) with a shared 2MB L2 cache. Each core and it's private cache constitute a separate voltage/frequency domain such that DVFS can be applied independently on each domain. There are 9 Voltage and frequency levels ranging from 1.55V-0.8V and 2.66GHz-1.6GHz respectively [31] . Voltage and frequency level for uncore i.e. L2 cache and memory, is fixed to 1GHz and does not change with DVFS. We take a control period of 3ms, as discussed in section 3, which is well within the scheduling range of the OS and can allow it to make DVFS changes to the core. Thus we don't require a hardware DVFS handler. We set the DVFS transition latency to a fixed 1us.

We use gem5 [4] full system simulations with Out of order processor cores and classic memory model. Linux version 3.13.0 is the OS we run, with thread migration disabled and each thread is pinned to a different core to simplify the analysis and get rid of complications due to context switching. For modeling power and energy, we used Cacti [22] (version 1-6.5) and MCPAT 1.0 [21] with process technology of 32nm.

Benchmark applications are taken from widely used parallel workload suits SPLASH-2 [33] and PARSEC 3.0 [3]. These benchmarks use Pthreads and OpenMP thread libraries to handle synchronization, which are instrumented to add pseudo

ISA	ARM version 7
no. of cores	4
core type	4-wide out-of-order
L1 D-cache	64 KB, private, 2 cycles
L1 I-cache	32 KB, private, 1 cycles
L2 (last level)	2MB, shared, 12 cycles
DRAM	512MB, 100 ns
Core DVFS states	voltage: 1.55V - 0.8V
9 levels	frequency: 2.66GHz - 1.6GHz
Control period	3ms

Table 5.1: Hardware configuration for simulations

instructions to mark the start and end of the spinning phases due to locks, barriers and conditional variables. The applications are cross-compiled for ARM with GCC ARM Embedded tool chain version 4.7.3. We choose a subset of applications for our analysis as shown in table 5.2, which represents different kinds of scalability behaviour as is evident from our discussion in section 1.2.

Suite	Benchmark	Input	Scalability variation
PARSEC	Bodytrack	simmedium	low
	Fluidanimate	simmedium	high
SPLASH-2	fmm	simmedium	low
	lu_ncb	simmedium	medium
	ocean_cp	simmedium	high

Table 5.2: Benchmark chosen for analysis

5.2 Evaluation

In this section we present the performance, power and energy analysis of PCSLB for benchmarks shown in 5.2 and compare them with the best performaing DVFS policy, Max_Bips from [15].

5.2.1 Prior work: Max_BIPS

Max_Bips is the best performing DVFS policy as presented in [15]. It finds the optimal combination of DVFS operating levels for the cores on chip using an exhaustive search and predicts the power and BIPS (Billion Instructions Per Second) values for all the possible combinations of DVFS levels. This helps in deciding the optimal operating point such that we get the best possible performance while in the limits of power budget. We implemented Max_Bips as shown in algorithm 2. In order to compare our results, we fixed the power budget for max_bips to the power usage of PCSLB and compared the performance degradation and the energy savings.

5.2.2 Results

Figures 5.1-5.3 show the performance, energy and power comparison in percentages as compared to "upper bound". We define "upper bound" as the system with all cores running on F_{max}. From figure 5.1, it can be seen that the performance degradation for PCSLB is less than 0.5% for lu_ncb, fluidanimate and fmm while it is around 2% for ocean_cp and bodytrack. This shows that the load balancing algorithm is trying to achieve the balance in an accurate way with minimal performance loss due to prediction errors. For benchmarks which have high scalability variation, fluidanimate and ocean_cp in our case, we get high power and energy savings. For fluidanimate PCSLB gets 8% and for ocean_cp it gets close to 11% energy savings which is 16X and 5X more than their performance degradation respectively. For

Algorithm 2 Max_Bips Implementation

```
Require: End of Control Period        ▷ Run the algorithm after every control period
 1: for i:= 1 to Number of threads do
 2:     $BIPS_i \leftarrow$ BIPS value for this period
 3:     $POWER_i \leftarrow$ Power usage for this period
 4:     $f_{current} \leftarrow$ Frequency current thread is running on
 5:     for all f in all frequency levels do
 6:         $BIPS[f][i] \leftarrow (\frac{f}{f_{current}}) * BIPS_i$
 7:         $POWER[f][i] \leftarrow (\frac{f}{f_{current}})^3 * POWER_i$
 8:     end for
 9: end for
10: Choose the frequency of each core such that the total power of the chip is below the
    power budget and the BIPS is maximized.
```

lu_ncb and fmm it gets 2% energy savings which is 4X more than their performance degradation. These benchmarks are pretty scalable as evident from figure **??**. So PCSLB chooses F_{max} for the cores most of the time. Bodytrack is also very scalable as shown in figure **??** and there are a lot of spikes in scalability which forces PCSLB to make some wrong dynamic decisions even after considering the exponential moving average as explained in Algorithm 1. This causes bodytrack to show more performance degradation than the energy savings.

As it can be seen, PCSLB always performs better than Max_bips for two major reasons. Firstly, Max_bips tries to maximize BIPS (Billion Instructions Per Second) and as we have shown in section 3, committed instructions per cycle which is in a way equivalent to BIPS, is not a significant performance predictor when it comes to parallel programs as they can have many dynamic instruction due to spinning

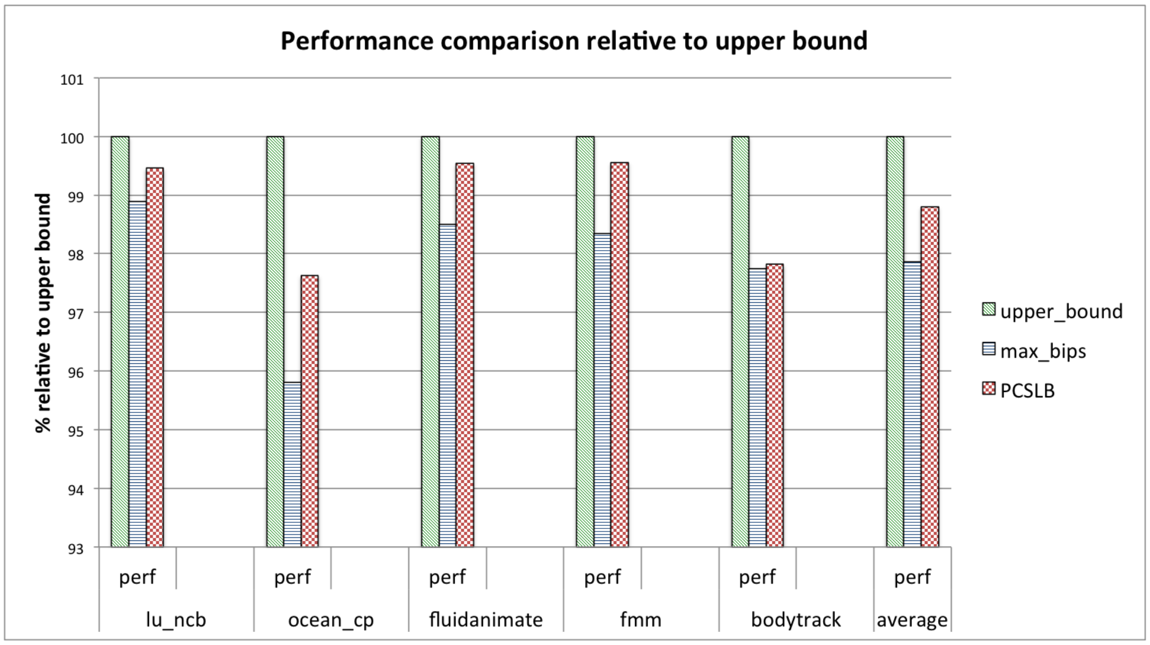

Figure 5.1: Performance degradation in % for PCSLB and max_bips as compared to upper_bound

which does not contribute to a thread's performance. So making DVFS decisions based on BIPS is not an efficient way. Secondly, it doesn't have any knowledge of the load imbalance in the program which can cause it to scale the frequency of a non critical thread which will not give much performance benefit. Max_bips shows similar power savings to PCSLB as we fixed the power budget to the power usage of PCSLB. Other than Bodytrack, Max_bips show more performance degradation and less energy savings than PCSLB.

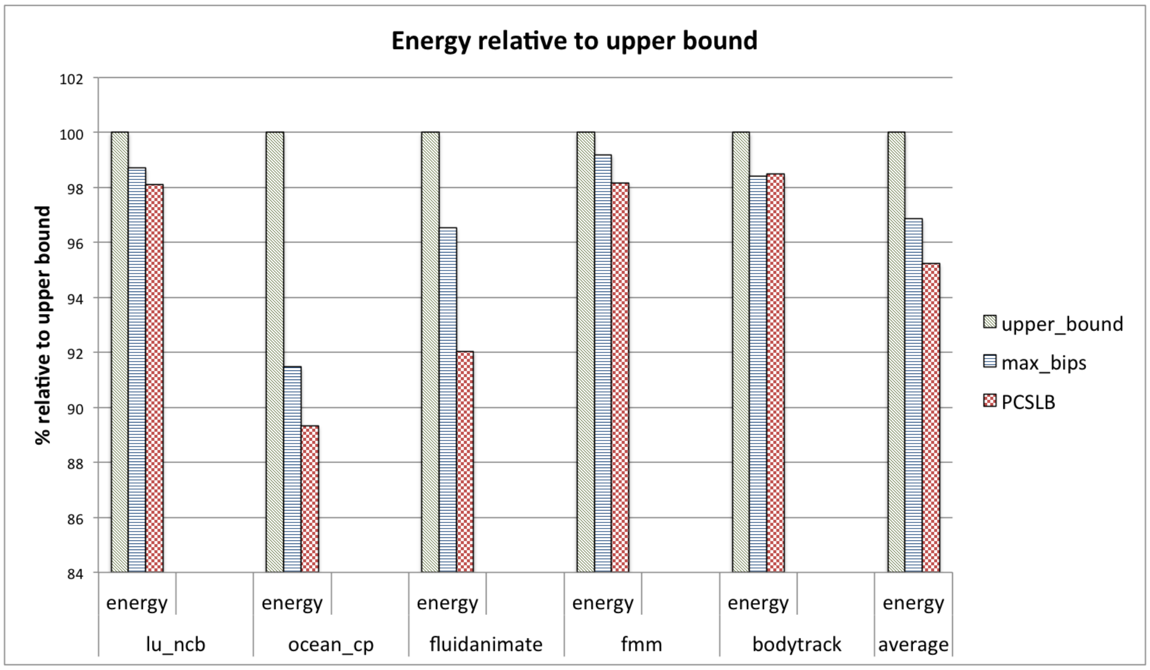

Figure 5.2: Energy savings in % for PCSLB and max_bips as compared to upper_bound

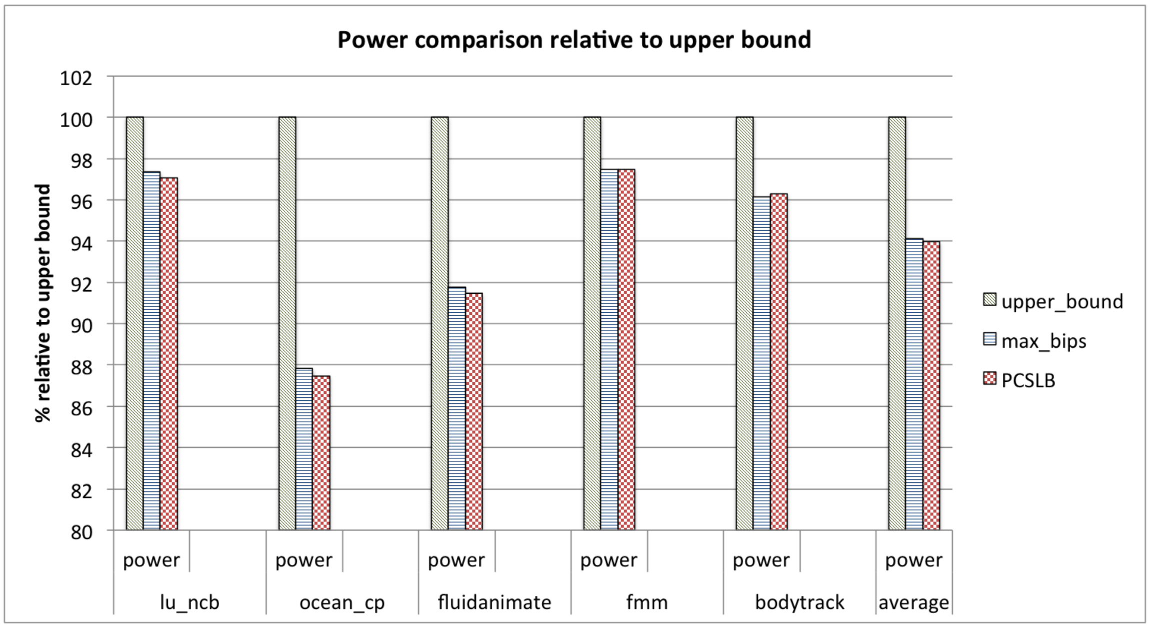

Figure 5.3: Power savings in % for PCSLB and max_bips as compared to upper_bound

6. CONCLUSION AND FUTURE WORK

Parallel programs have an inherent load imbalance problem due to synchronization overhead which becomes a bottleneck and limit the performance and energy benefit achieved by the thread level parallelism. To solve this, one needs to detect the load imbalance and scale frequency of individual threads accordingly. But detecting load imbalance alone is not an accurate and energy efficient solution as it does not take in account the scalability of the thread, i.e. the impact of DVFS scaling on thread's performance. This causes the algorithm to make wrong DVFS decisions for non_scalable phases, which are oblivious to frequency changes. So adding the scalability information to the algorithm can provide huge energy savings and an accurate load balance. In this work, we first model scalability as a function of available hardware monitors for parallel workloads using static regression statistics. We then use this model along with thread criticality prediction to devise an accurate load balancing algorithm, PCSLB. We show that we can achieve energy savings more than 10% with very minimal performance degradation of less than 1% in most cases, as compared to the upper bound where all threads are run on the highest frequency. This proves that we can get an accurate load balance when we combine thread criticality prediction with each thread's scalability information.

For this work we have considered a 4 core machine and widely used parallel benchmarks, SPLASH [33] and PARSEC [3]. Future work includes working with 8 and 16 cores to see how much more benefit we can achieve. We also wish to work with other memory intensive benchmarks, as SPLASH and PARSEC don't have many applications with high scalability variations. As a future work, we would also like to experiment with more predictor variables to see if we can model scalability in a more accurate way.

REFERENCES

[1] 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.

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

[3] C. Bienia, S. Kumar, J. P. Singh, and K. 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.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[5] 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 *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 240–249, New York, NY, USA, 2008. ACM.

[6] K. 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] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. *SIGARCH Comput. Archit. News*, 41(3):511–522, June 2013.

[9] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. 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.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

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

[13] E. Grochowski, R. Ronen, J. Shen, and H. 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.

[14] 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.

[15] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. 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.

[16] 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.

[17] 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.

[18] G. Keramidas, V. Spiliopoulos, and S. 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.

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

[20] 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.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[22] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *ICCAD: International Conference on Computer-Aided Design*, pages 694–701, 2011.

[23] C. Liu, A. 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.

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

[25] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *SIGARCH Comput. Archit. News*, 31(2):14–27, May 2003.

[26] R. Miftakhutdinov, E. Ebrahimi, and Y. 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.

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

[28] 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*, pages 1–8, July 2011.

[29] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. 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.

[30] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. 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.

[31] 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.

[32] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.

[34] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGOPS Oper. Syst. Rev.*, 38(5):248–259, October 2004.