DYNAMIC FEEDBACK-DRIVEN THREAD MIGRATION FOR ENERGY-EFFICIENT EXECUTION OF MULTITHREADED WORKLOADS

by

Claudia Alvarado, B.A., B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2014

Committee Members:

Apan Qasem, Chair

Dan Tamir

Martin Burtscher

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I dedicate my master thesis to my family, especially my mother and my sister, and to my closest of friends. Their emotional, moral and financial support made this all possible.

ACKNOWLEDGEMENTS

I wish to thank my committee members for the years of support and mentoring that they have each provided me with, offering me invaluable opportunities for growth and improvement. I would especially like to thank Dr. Apan Qasem for his unwavering patience and constant support throughout this entire process. I would also like to thank Dr. Dan Tamir for setting an unprecedented example in work ethic and always encouraging me to challenge myself. Last of all, I would like to thank Dr. Martin Burtscher for taking the time to turn his profession into a craft, and for inspiring me to never settle.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

Multicore architectures require sound thread to core mapping policies in order to exploit the efficiency and parallelism that multi-threaded programs offer. Traditionally, the operating system scheduler focuses on temporal aspects of performance such as execution time and latency, disregarding other factors that may have significant impact on the system. For example, judicious thread migration decisions can provide significant power savings. Typical schedulers, however, fail to make power aware migration. This master thesis focuses on comparing the effects of using resource aware analytical models, and a machine learning model, on making power aware thread migration decisions.

The first analytical model uses a greedy algorithm, and aims to balance the load on processors, based on each core's utilization level, via thread migration. We use a novel approach to derive core utilization levels, utilizing the dynamic feedback provided by performance counters, as well as a modified utilization metric that more accurately reflects the state of a processor.

The second analytical model is aware of the processors' shared resources and aims to reduce any contention via thread consolidation. Hardware performance counters that reflect high miss rates in certain shared resources are evaluated and compared to established thresholds; the model then recommends to either consolidate or preserve the default scheduling. The new affinity configuration, if any, is expected to promote greater power savings.

The last evaluated model uses a machine learning approach to recommend a final affinity configuration for a workload at runtime. The novelty of this approach again lies in the utilization of hardware performance counters for model training. A total of four metrics derived from a subset of available counters comprise the feature vector of the model. Three algorithms are employed with the model: a decision tree to aid with visualization, a support vector machine which provides a categorical approach, and the statistically based Bayesian model.

The three models are evaluated with various single and multi-program workloads, where each workload differs in certain tunable parameters such as initial affinity configuration or thread count. Results reflect differences in execution times when applying the models and when utilizing the default OS scheduler. Additional comparisons in power consumption reveal strengths and weaknesses of each approach, and a final evaluation recommends the most beneficial approach for preserving power.

CHAPTER 1

Introduction

There is a continuously growing demand in the computer industry for faster processors to perform increasingly complicated tasks. One major implication of increasing a processor's operating frequency is an increase in power consumption resulting in heat dissipation. This design challenge is mitigated with the introduction of multi-core processors, which offer a cost effective way to meet performance requirements while minimizing the inevitable growth of the hardware's footprint. Software, however, is needed to maximize the benefits that multi-core architectures offer and plays an important role achieving high performance. Multithreaded applications increase performance by concurrently running tasks or threads for workloads. In order to fully harness the multi tasking capabilities of concurrent threads, an additional software solution is needed to extract and exploit the parallelism. Furthermore, multi-program, multithreaded workloads must be managed carefully, particularly in regards to shared resources. This management can be handled by thread scheduling policies that not only aim to increase performance, but also attempt to reduce power consumption and are aware of shared resources.

Because thread management is so important for multicore architectures, many techniques for thread placement, migration and scheduling have been incorporated into current operating systems. However, these techniques are sub-optimal because they do not consider power. Linux, for example, uses a timesharing and priority based scheduling policy that classifies processes as I/O Bound or CPU Bound, and schedules them accordingly. The main goals of the Linux scheduler are to avoid starvation, give

priority to interactive applications, scalability, and reducing idle time [1]. Thread migration is a technique commonly used by OS schedulers to encourage parallelism, however, scheduling decisions are made by focusing on evenly distributing work among the cores typically disregarding power consumption and shared resource ramifications. The Linux scheduler fails to consider some influential factors; it follows that the most optimal scheduling algorithm would require a vast increase in the considerations taken when making migration decisions.

One way to supply more information to the OS is to take advantage of hardware performance counters or performance monitoring units (PMUs). PMUs can measure a variety of performance and power related metrics such as instruction count, shared resource miss rates, and energy and power consumption levels. Although PMUs can provide a lot of information and are routinely used in performance tuning, their use in OS-specific tasks is limited. This is partly due to sampling overhead, and partly due to the difficulty of modifying OS kernels.

This thesis develops several strategies for mapping and migrating threads on multicore systems for improved energy savings. The intention is to implement three different migration models as part of the operating system scheduler. However, in order to enable efficient testing, experimentation, and evaluation, the algorithms are developed and implemented in the Linux user-space. The novelty of the proposed strategies lies in the fact that all three models can be implemented in user-space, with no modification to the kernel and that all three models make extensive use of PMUs to gain insight into workload behavior.

The main contributions of this research are as follows:

1. A user-space software system for measuring and utilizing PMU events in OS-specific tasks such as thread scheduling and mapping
2. A load balancing algorithm for maintaining power caps in multi-threaded, multi-programmed environments
3. A thread migration policy that leverages PMU readings to reduce resource contention
4. A machine learning strategy that makes thread migration decisions based on workload characteristics to provide better energy savings.

This thesis paper is organized as follows. Chapter 2 provides a conceptual background and discusses traditional operating system scheduling methods. A description of power aware and resource aware migration policies and a review of performance metrics define new contributing factors that are considered in the proposed migration policies. Chapter 3 outlines relevant works that focus on operating system power management techniques. The literature review reveals that closely related work fails to extend power management policies to larger workloads. This research aims to evaluate the effectiveness of power aware migration techniques on larger workloads in an effort to simulate a more realistic working environment. Chapter 4 presents descriptions of supplemental tools that were utilized in workload generation, the developed power aware and resource aware migration policies, and the machine-learning algorithm. Heuristics and development stages are highlighted, as well as variations that were explored and abandoned. Chapter 5 provides a detailed description of thread migration strategies. The derivation of metrics for each analytical model is described, followed by the developed policies. A review of each of the three models, Decision Tree, Support

Vector Machine and Bayesian Model, used with the machine learning approach follows along with a description of the training data generation process. Chapter 6 presents the experimental setup, results and evaluation. First a review of the platforms that the models were tested on is presented, followed by the benchmarks used and a description of the generated workloads. An evaluation of each model is made by comparing the power consumption or execution times of workloads scheduled with the OS scheduler and with each proposed model. The effectiveness of the machine-learning algorithm is assessed by illustrating a comparison of each workload's power savings to power consumption levels of workloads that are traditionally scheduled. Additionally, execution times are evaluated to determine the benefit of each model individually, and the greatest gain achievable given a combination of all three models. Chapter 7 provides an overall synopsis of the problem, the proposed solutions, and main contributions of the research. Emphasis is placed on the fact that significant power savings were successfully attained through power aware thread migration policies. Suggested options for expansion of the presented research conclude this paper, noting additional possible benefits that may be achieved.

CHAPTER 2

Background

This chapter provides a description of fundamental concepts concerned with operating system scheduling multi-threaded applications in a multicore environment.

2.1 Multithreaded Applications

A *process* is an instance of a program currently in execution, which defines the address space, and contains a program counter, a stack pointer, and any opened file handles. Every process requires access to certain resources such as memory, CPU time, files, and I/O devices, and must be in the 'ready' state to be considered for execution. A *thread of execution* is the smallest sequence of instructions that can be managed and must live within a process. A thread shares the address space with its parent, and must reside within a process. Each process and thread is identified with a unique process id (PID) or thread id (TID), respectively. A *multi-threaded process* consists of a number of concurrently executing threads that are each spawned by the parent process. Threads running on a single core share computing units, CPU caches, and the translation look aside buffer. A process or thread must be in one of five states during its execution: new, ready, waiting, running, or terminated. Figure 1 shows the state diagram for a process and illustrates the transitions between each state.

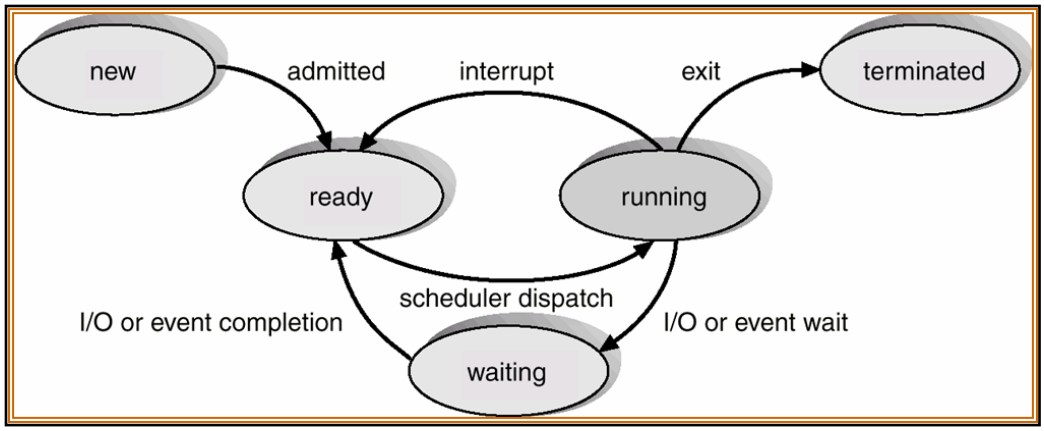

Figure 1: Process state diagram

2.2 Thread Scheduling and Migration

It is when a process or thread is in the ready state that the operating system can schedule the task to run on a processor, referred to as *task scheduling*. The mapping configuration that each thread has for a particular core is called the thread's *affinity*. Schedulers aim to optimize specific metrics such as reduced core idle time, or increased performance and take different factors into consideration. *Power aware task scheduling* considers a task's power consumption and aims to create a mapping, which reduces overall system power use. *Power* is defined as the rate of energy measured in Joules per second or Watts.

Another facet of task scheduling is *task migration*, which reassigns currently running tasks to different processors.

2.3 Linux Scheduler

Linux provides *preemptive multitasking* where the scheduler is responsible for deciding when to cease and resume a process. A *time slice* is a predetermined amount of time that a process runs prior to being involuntarily stopped and is dynamically

determined. The basic data structure of the Linux scheduler is a *run queue* for each core, which holds all of the runnable process for that core and uses the spin lock method for access. Each run queue is comprised of an expired and an active priority array. Each priority array has one queue of runnable processes per priority. Linux uses a fast find first search algorithm to search a priority bitmap for the highest priority runnable task in the system. Figure 2 illustrates the data structures that the Linux scheduler is built on [2].

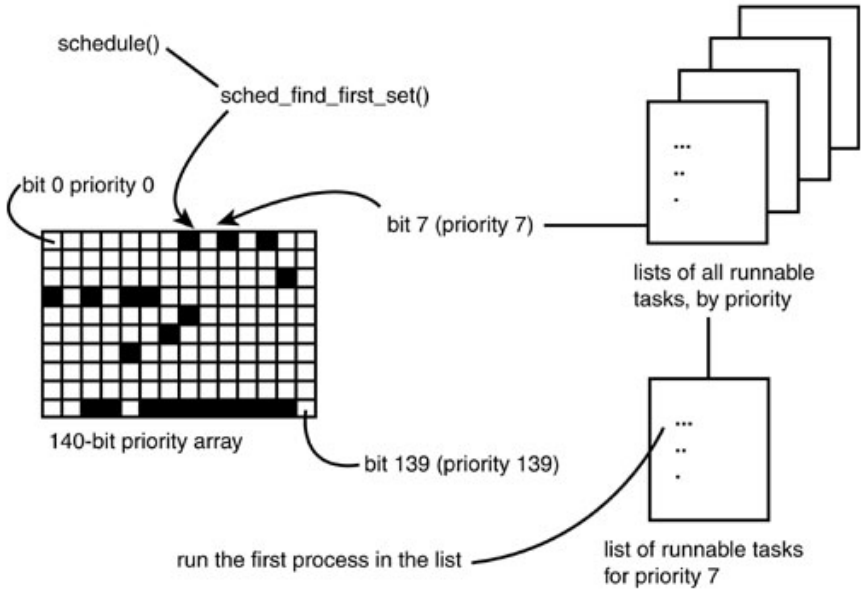

Figure 2: Linux process scheduler data structures

Linux gives priority to highly interactive tasks by reinserting them into the active priority array rather than the active array. In addition, a load balancer is employed to maintain run queue population.

2.4 Load Balancing

Load balancing is a power saving technique that evenly distributes tasks or processes between the processors in order to maintain a balanced system where cores are somewhat equally utilized. Linux implements a load balancer in the scheduler that aims

to balance run queues and is invoked in one of two ways. The system balance is checked at a fixed time interval, and if at any point there is a core with an empty run queue or if there is an imbalance between the run queues, the balancer is utilized.

CHAPTER 3

Related Work

This chapter provides a survey of related research. Research that employs the use of performance monitoring units (PMUs) is first discussed, followed by work exploring the benefits of processor load balancing. Next is an in depth study of various thread scheduling and migration techniques. The review is concluded with a study of closely related work that emphasizes power aware scheduling.

3.1 Hardware Performance Counters in Performance Tuning

Chip manufacturers first started to expose PMUs to software in the late 1990s. The PMUs were increasingly utilized in research regarding performance tuning and code optimization. Eranian was one of the first to describe the utility of PMUs in improving application performance [3]. Subsequently, researchers have employed PMUs in automated performance tuning [4, 5], performance modeling and recommendation of transformations [6], optimizing for the memory hierarchy [7], better resource utilization [8], estimation of processor temperature [9] and power consumption [10, 11], and DVFS-based scheduling.

Although PMUs are highly useful for tuning high performance computing (HPC) applications, their incorporation into OS-based strategies is still very limited. One of the most notable utilization of PMUs in OS work is by Azimi et al., where feedback from hardware performance counters is used to dynamically partition the cache for workload execution [12]. Bhattacharjee et al. exploit the use of performance counters to determine which threads in a multithreaded process are the critical threads [13]. The motivation is that when the OS is able to identify a critical thread, actions can be taken to expedite its

processing, such as increasing frequency, allocating on chip resources, or load balancing. Banikazemi et al. develop a user-space meta scheduler that provides feedback regarding resource congestion in cores to the OS [25]. Their strategy results in a 14% overall improvement on the SPEC CPU workload. The considered metrics, however, do not capture key sharing characteristics, nor are multi-threaded workloads evaluated.

3.2 Load Balancing

Sarood et al. explore the effects of temperature aware load balancing on a large scaled system consisting of 128 cores [14]. The motivation is that cores running at high frequencies require a certain amount of cooling, which can often consume up to 50% of the available power. They establish a migration policy that focuses on reducing each core's temperature while minimizing any total runtime penalty, and consequently lowering the overall power consumption. They do not extend the evaluation to include smaller systems consisting of a more readily available core count.

Musoll explores the power saving benefits of load balancing by clustering core groups rather than individually gating cores [15]. The clusters are then balanced and power gating is implemented on a cluster level. Results indicate that the methods employed yield a low overhead, and increase the overall reliability of the processor. We propose that a similar methodology can be applied to individual cores, and yield comparable power savings once a system is balanced.

3.3 Thread Placement Strategies

The amount of literature focused on thread placement and scheduling techniques within the OS is large. User-level approaches, however, are relatively few. Among the

developed user-level techniques capable of affinity management, most are limited to single applications or single-program workloads.

Recently, Zong et al. have proposed two algorithms for scheduling parallel applications on large clusters [16]. Their framework takes a precedence-constrained task graph of the application to be scheduled as input, and emits a schedule predicted to be the most energy efficient. Tedorescu and Torellas present a power management algorithm that considers voltage and frequency variations among the cores and attempts to improve performance within a given power envelope [17]. Linear programming is utilized to implement the algorithm, and it is intended to complement the existing OS-scheduling policies. Brown et al. explore methods to partially copy data and pre-fetch instructions, known as their '*working set prediction*', in to the new cache [19]. Results show that implementing the latter policy directly caused up to a double increase in performance for short-lived threads, in addition to speculative multi-threaded environments. Experiments were not extended to include entire workloads, however, and adopting the methodology to the proposed research would reveal the power saving benefits of considering locality when making thread migration decisions. Chen et al. compares two scheduling policies: work stealing which is a traditional design utilizing a double-ended queue, and parallel depth first (PDF) which is designed to promote concurrency between threads which have a large overlapping working sets, also known as constructive cache sharing [21]. PDF specifically, reduces the overall amount of cache needed, and consequently the amount of power required.

Tam et al. consider locality and shared resources when scheduling on a multiprocessor chip [18]. They create a scheduling algorithm that takes into account

whether or not concurrently running threads will be attempting to access the same data. Threads that do share data are mapped to either the same or a closer core so that latency is minimized. The pattern detection that is used to determine if resources are shared introduces a negligible overhead as a consequence of dynamically employing performance counters that are easily accessible. Once the data has been analyzed and sharing patterns have been detected, selected threads are clustered together based on the patterns, and are scheduled to be placed on the respective cores as close to each other as possible. Thread migration decisions are made with the ultimate goal of having a balanced system on the chip level, not the core level. We propose a scaled approach that takes a closer look at individual core characteristics and aims to balance the system with a finer granularity. In addition, rather than using pattern detection to identify opportunities to exploit locality, we rely on the dynamic feedback of performance counters. Kandemir et al. studies the effects of exposing the system's memory topology to a scheduling algorithm [20]. A comparison is made between two base cases where scheduling is handled without modification, and with localization optimization techniques employed, such as loop permutation, which permits modifying execution order, and blocking, which consolidates sections of code to promote data reuse, respectively. Results show that the algorithm aware of the memory topology improved execution time. However, power consumption was not reported. We hypothesize that a similar behavior will be seen in regards to power consumption when a power aware scheduling algorithm is aware of the memory topology of a system. Merkel et al. develop heuristics that schedule threads based on shared resources, and couple this approach with DVFS based techniques [22]. They evaluate their strategy on a workload

with homogenous sharing patterns, and show that their strategy is able to significantly reduce the Energy Delay Product (EDP). Boyd-Wickizer et al. propose a technique that operates at an object level [23]. The objective is for inter-core thread migration decisions to be based on data structure access, bringing threads closer their data, thereby reducing memory latency.

Singh et al. develop a user-space meta scheduler [11]. In their work, hardware performance counters are used for estimating processor power consumption. Although multithreaded benchmarks were used for evaluation, results were only reported for processes spawned with one thread. Additionally, rather than providing a migration policy, the proposed scheduler suspends and resumes entire applications to keep each processor running under an established envelope.

Work published by Vega et al. is most closely related to this research. Vega proposes a thread consolidation technique that focuses on increasing power efficiency without sacrificing performance for multi-threaded workloads [26]. Consideration is given to the asymmetrical properties of software and hardware threads and is exploited; resulting in increased power efficiency and yielding benefits from core gating in addition to increasing throughput. Vega's analysis shows that there is a power-performance trade off that is more beneficial for applications with poor performance scalability, and further concludes that the aforementioned trade off is affected by micro-architectural details in addition to application affinity and core count. These two factors, however, although mentioned, are not actually utilized to implement a power saving policy. Vega et al. extend work from the 2013 publication with an implementation of the aforementioned proposed policy [27]. The implementation exploits the use of a thread consolidation

heuristic as well as core gating to increase power efficiency. Research was focused on running single applications at a time and analyzing their performance and power consumption based on a number of varying configurations. We propose to create a similar environment for an increased number of concurrent applications, and evaluate the performance and power consumption of multi-application workloads rather than a single, multi-threaded application.

CHAPTER 4

Metasched Framework

This section focuses on the implementation details of the overall framework, called Metasched. Figure 3 outlines the main component tools of Metasched and shows their interconnections. Our framework takes advantage of a number of userspace utilities that come standard with current Linux distributions. The components of Metasched can be used as standalone tools or in an integrated manner (as is done for the work presented in this thesis). In the following sections, we provide brief descriptions of the main components of Metasched.

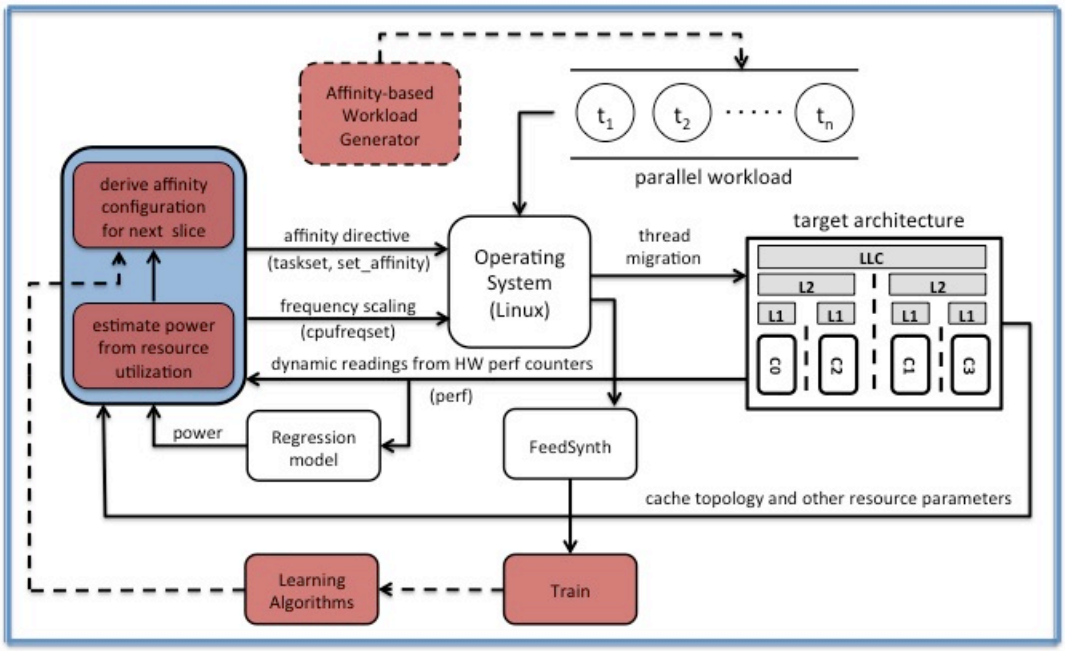

Figure 3: User space algorithm exploration framework

4.1 Workload Generation

The workload generator can launch a set of programs from a pre-defined list using a variety of configurations. This tool be can be used to create multi-program and multi-threaded workloads, exhibiting different characteristics. The available options include:

number of applications to run, launch interval, number of threads for each process, initial thread affinity, and data set size. Being able to fine-tune all of the parameters simplifies the analysis and allows for changes in individual components to be easily identified. In this work, the workload generator is used to create multi-threaded workloads from selected applications and kernels from the PARSEC benchmark suite.

4.2 FeedSynth: Feedback Collector and Synthesizer

Current architectures expose a large number of hardware performance counters that can provide significant insight into application performance. These counters can be probed in software with tools that are standard in most Linux distributions. Examples of such tools include perf, likwid[28], and HPCtoolkit[29]. FeedSynth provides an interface to perf and facilitates the reading of PMU registers directly in our system. Feedsynth is capable of gathering information from all exposed counters on current AMD and Intel processors. In this work, we only utilized a subset of theses counters, as described in Chapters 5 and 6.

4.3 Migrator

The thread migrator module in Metasched dynamically changes the affinity of running processes using the Linux *taskset* utility. *Taskset* sets hard affinities for a process, which implies that the affinities set by *taskset* are always honored by the OS. The thread migrator tool extends the capabilities of *taskset* and facilitates the setting of affinities at the individual thread level. In our work, *taskset* is also used to probe the current affinity of running processes. The thread migration and mapping algorithms described in this thesis are all implemented within the thread migrator module.

Each migration model provides a power aware thread affinity configuration for any given workload. The modules are each invoked by first specifying a number of parameters: workload generator, number of programs in workload, a starting affinity, the number of threads allocated to each program, and the dataset size. The model choice dictates which hardware counters are sampled and used as dynamic feedback throughout the execution of the workloads to ultimately guide the thread migration decisions.

4.4 Power Estimator

To perform any type of power-aware migration, mapping, or scheduling, the algorithms require feedback in terms of the amount of power that is being consumed. Although some current architectures provide counters that can be probed to get an estimate of processor power consumption, there are still many platforms where such counters are not supported. To be able to implement thread migration and scheduling techniques on these platforms, Metasched includes a regression-based model for processor power estimation. The power estimator, named WattsUp, estimates processor power consumption in Watts, using four different PMU events.

4.5 Frequency Scalar

Metasched also supports DVFS-based optimization for power and energy. This module changes processor states from user space by directly modifying the *cpufreutil* system files on Linux systems. The number of transitions depends on the number of supported frequency states on the target architecture. We do not make use of the DVFS-module in the context of this research.

CHAPTER 5

Thread Migration Strategies

This chapter provides the motivation behind and a detailed description of each thread migration strategy. Heuristics are discussed for the load balancing and shared resource models, and details of the machine learning algorithm concludes the chapter.

5.1 Load Balancing for Power Caps

Power spikes can cause significant increase in energy costs for large data centers. To avoid these spikes, it is important for data centers to operate under a given power budget. Techniques such as frequency and voltage scaling (DVFS) and core gating can be used to ensure that power caps are maintained on such systems. In this work, we devise a load balancing strategy to accomplish this goal. We select load balancing because it is known to be effective in maintaining power caps with the least amount of performance penalty. The novelty of our load-balancing algorithm comes from the use of metrics derived from PMU events, and in its ability to operate from a user-space without any OS intervention.

5.1.1 Core Utilization Metric

To effectively balance the load on processors, the operating system must determine the processor utilization levels across the system. However, it has been shown that CPU utilization, the conventional metric used by operating systems, does not capture a multicore platform's utilization levels accurately. In this work, rather than using the conventional CPU utilization metric used in current operating systems, we derive a more accurate and descriptive metric for core utilization derived directly from hardware performance. This metric can gauge utilization levels not only at the processor level, but

at core levels (both physical and logical) as well. The core utilization metric is derived as follows:

Core Utilization = CPU time / Elapsed Time
where,
Elapsed time = length of time slice
CPU time = UNHLTED_CYCLES * clock cycle time
where,
clock cycle time = 1 / clock rate

Furthermore, we determine that the core utilization metric provides a reasonable estimate of per-core power consumption. This conclusion is particularly applicable for power-aware optimizations where power counters are either not available, or require a significant amount of overhead to estimate. On such systems, the models described in this paper can be implemented using core utilization levels as the objective metric. As we demonstrate in Chapter 6, using the core utilization metric for such power-aware schemes, provides similar benefits as measuring power directly.

Figure 4 shows the core utilization and power consumption of two example workloads. The first workload shown on the left requires high levels of computation and the second workload exhibits high levels of cache sharing. In both cases it is evident that although the magnitude of the values are different, there is a high correlation between core utilization and power consumption. Hence, although core utilization cannot be used to produce accurate estimates of actual power, the metric is useful for guiding optimizations.

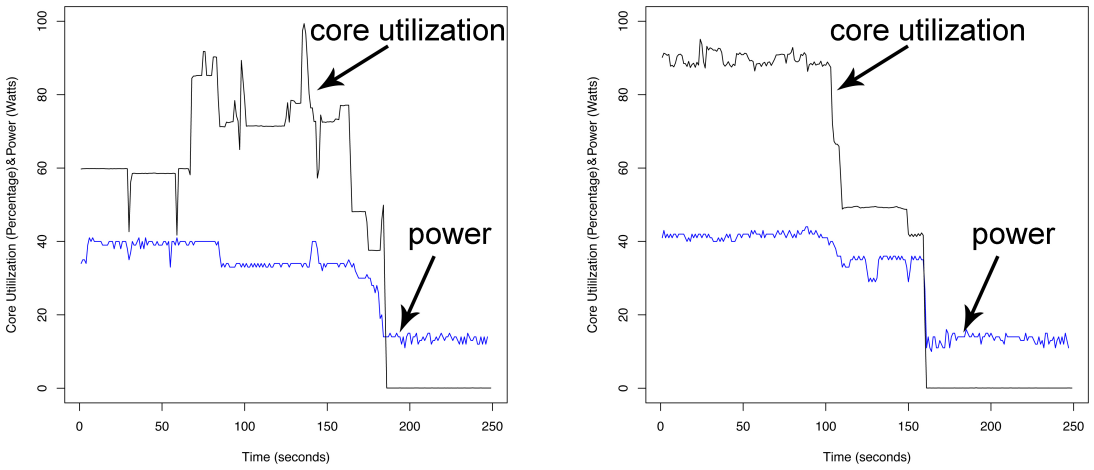

Figure 4: Core utilization and power consumption correlation

5.1.2 Balancing Algorithm

The first analytical model is founded on the idea that core utilization levels are highly correlated with power consumption, and that a system must operate within an established power threshold or cap. As a result of the established relationship between power consumption and core utilization, the first model uses a greedy approach and aims to maintain a balanced state for the system to reduce overall power consumption. The implementation evaluates the core utilization level metric at each time slice during the execution of a workload. Consideration is given to the highest and lowest utilized cores. When the difference in their levels is above a predetermined threshold (reported results reflect a threshold of 25%), the system is considered unbalanced. The algorithm then modifies the affinity of threads running on the highest utilized core to also run on the lowest utilized core. We call this increasing the affinity of a thread. We opt for increasing the thread affinity rather than actually migrating the threads to reduce the possibility of load imbalance in future time slices. A reevaluation of performance counters on the next time slice reflects the impact of the thread migration, if any. If the

level of imbalance between any two cores is below the threshold, then the current affinity or thread mapping is preserved. The performance counter evaluation and thread migration decisions are repeated until the workload has completed execution or the algorithm is terminated via an external signal.

Initial model implementation developed a migration policy for PIDs rather than TIDs. As a result, if a program was launched with *n* threads and a migration decision was made, migration of the PID would effectively migrate all of the *n* TIDs belonging to that PID. The result was an ineffective model that lacked the granularity needed to truly promote a balanced system. We augmented the model to focus on migrating individual threads rather than entire processes, facilitating the maintenance of a balanced load on all available cores. Figure 5 illustrates the overall logic flow of the described model.

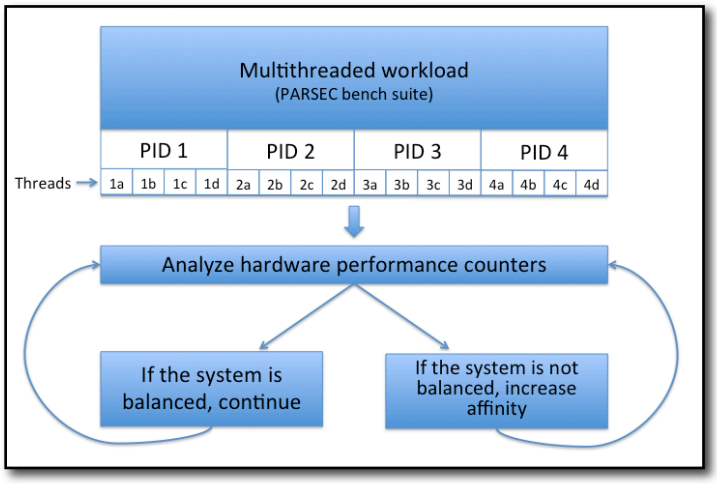

Figure 5: Load balancing analytical model logic flow

5.2 Thread Migration for Shared Resource Utilization- Feature Selection

5.2.1 Deriving Relevant Metrics

Hardware performance counters offer a significant amount of information about program behavior and system state. This is particularly helpful for dealing with

multithreaded applications due to a number of contributing influences on overall performance. For example, 656 events can be monitored on the Intel Sandy Bridge platform, which in turn can be used to derive metrics to characterize specific aspects of workload performance. These derived metrics can also be used to construct suitable features for machine learning based optimization strategies. Monitoring all of the available counters at once poses some issues, however:

1. The features considered by a machine-learning algorithm should include only the necessary, representative features to prevent diminishing prediction accuracy.
2. Architectures limit the number of counters that can be simultaneously probed at runtime. Consequently, analytical models using dynamic feedback to derive metrics must limit the number of counters probed at once. Additionally, a machine learning algorithm utilizing counters as features is also limited and must adhere to what the architecture supports, which is traditionally four to eight counters.

To accommodate the counter-probing limit and establish the relevant counters, we develop a set of synthetic micro-benchmarks designed to utilize various shared resources on the target platform. Each program is run with varying affinity configurations and different levels of resource utilization, which are controlled via a set of command line parameters. On each run, a large subset of available counters is sampled. On the Intel Core 2 Quad platform, 68 counters are considered, and 120 counters are looked at on the SandyBridge platform. This initial subset selection is founded on expert knowledge and is performed to avoid sacrificing experimentation time. However, the strategy can be extended to sample all available counters.

The initial subset of counters is further pruned by applying Spearman's Rank Correlation and outlier classification to obtain the counters most closely related with micro-benchmark performance variations. We then remove highly correlated features by applying Principal Component Analysis to the subset, and establish the following four features of interest for the Core 2 Quad architecture:

- Access to L2 cache lines in a shared state
- LLC load misses
- DTLB misses
- L1 Instruction cache stalls

To ensure that the features are representative for various types of programs, we normalize the counter values, and arrive at a feature vector with the four features as shown in Table 1.

Table 1: Features used in resource utilization and machine learning algorithms

Feature Name, Normalization Formula and Required Counters
Shared Access Rate (SAR) = Number of L2 accesses / Total number of accesses
LLC miss rate = (Number of L2 load misses / instructions) * 1000
DTLB miss rate = (Number of DTLB load misses / total loads) * 1000
L1 stall rate = Number of L1 stalls / iL1 fetches

The metrics in this feature vector require six hardware performance counters to be calculated, all of which can be probed during a single execution.

5.2.2 Migration Algorithm

Based on the set of metrics derived through statistical analysis, we develop an analytical model to improve shared resource utilization in multithreaded workloads. The

model is developed in two steps. The first phase considers only the private L1 cache. Although L1 access is extremely fast, the cache size is relatively small, and often requires additional memory accesses in the event of a large working set exceeding the cache's capacity. We adopt a heuristic similar to the one described in the Section 5.1. At each time slice, performance counters are evaluated to calculate each core's L1 miss rate. Focus is turned to the cores with the highest and lowest miss rates. If the difference in miss rates is above 10%, threads that are running on the core with the highest rate are mapped to also run on the core with the lowest miss rate. Early model evaluation revealed an ineffective migration strategy, and a need for additional metrics.

The resource aware model is extended to consider additional shared resources when making thread migration decisions. The motivation lies in the fact that minimizing resource contention between running threads increases performance by reducing execution time and potentially reduces power consumption. To achieve this, a thread consolidation migration technique is used to reduce power consumption and potentially increase throughput. The inherent nature of multi threaded programs results in some idle threads, and possibly idle cores. The goal of thread consolidation is to reduce the number of cores that a process is running on, thereby reducing the power consumption of the vacant cores. To achieve this, consolidating threads assigns them to core groups or cohorts, which share certain resources such as the LLC in our case. This technique goes through a rigorous test and must overcome an initial distributed affinity in some cases, in order to better evaluate its performance. A distributed affinity will schedule threads evenly across all of the available cores without considering shared resources.

Following the start of workload, at the second time slice, the four metrics listed in Table 1 are derived. The counters are sampled at the second time slice rather than the first to avoid capturing any non-representative startup characteristics. The derived metrics are compared to established thresholds in order to dictate the decisions made by the model with a heuristic designed for a single or multi program workload. The thresholds are noted as follows: sharedAccess_T1, sharedAccess_T2, dTLB_T1, dTLB_T2, iL1_T1, LLC_T1 and LLC_T2. Figure 6 provides an outline of the strategy used in making migration decisions.

```
Shared Resource Heuristic:

if (sharedAccess > sharedAccess_T1)
        consolidate
else
        if (sharedAccess > sharedAccess_T2)
                if (dTLB < dTLB_T1)
                        consolidate
                else if (dTLB < dTLB_T2)
                        if (iL1 > iL1_T1)
                                consolidate
                        if (LLC < LLC_T1)
                                consolidate
if (LLC < LLC_T2)
        consolidate
```

Figure 6: Shared resource migration policy heuristic

If the decision to consolidate is made, then the single program and all of its threads are migrated to a single cohort, or core group. In the event of a multi program workload, each program is consolidated to one cohort, and assignments are alternated accordingly between cohorts.

5.3 Machine Learning for Thread Migration

The last developed model employs a machine learning approach. The model is trained with metrics derived from hardware performance counters collected during a

number of workload executions and makes the decision whether or not to consolidate the running processes. The metrics for each run or workload execution are saved in a *feature vector* used as model input. The features used are the same as the derived metrics evaluated with the resource sharing model: *shared line access rate, LLC miss rate, dTLB miss rate, and iL1 stall rate*. Three different classes of algorithms are explored with the machine learning approach. The first algorithm is a decision tree and is selected to provide a visual representation of how a decision is being made. The second is a support vector machine, which is a categorical approach, and the third is a Bayesian network, which focuses on statistics to make predictions.

5.3.1 Machine Learning Algorithms

Decision Tree

The decision tree algorithm takes an approach that evaluates each metric individually. The basic structure of the tree is binary in nature and consists of nodes and the transitions between the nodes. Each node of the binary tree represents one metric in the feature vector, and the value of the metric in question determines which side the tree is traversed. For instance, call the root of the decision tree 'Metric 1'. If the metric derived from dynamic feedback at runtime that is being evaluated is >= *n%,* traverse down the right side, otherwise traverse down the left. A new metric or node is then reached which again must be evaluated in the same manner. Every leaf of the tree represents a final consolidation decision [30]. Figure 7 illustrates a simplified structure of a decision tree. Each node is labeled with a metric and each transition is labeled with a threshold for comparison.

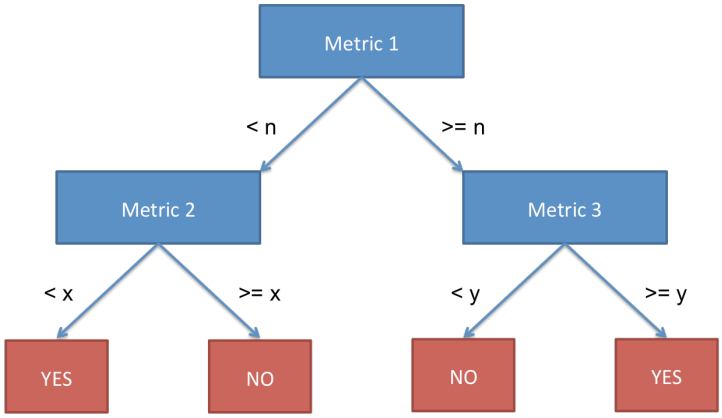

Figure 7: Simplified decision tree structure

Support Vector Machine (SVM)

The SVM algorithm is a binary model and focuses on complete 'scenarios' or sets of metric values in the input vector, and represents a categorical approach. The support vector machine features labeled examples that are generated through the training phase. Each example is comprised of the metrics derived for each of the workload runs. For example, consider an input vector consisting of two metrics *A*, and *B*. A simple scenario may be that if *A* > *50%* and *B* < *75%* then output = 0. Inversely, if *A* <= *50% and B* >= *75%* then output = 1. In our case, our input vector consists of the four aforementioned metrics. The thresholds, that observed values are compared to are learned by the model through evaluating a number of labeled experiments. The key to a more accurate model lies in creating a sufficient number of examples for the model to learn on so that there is more coverage of actual data that is to be evaluated once the model is invoked [31].

Bayesian Model

The Bayesian Model uses a statistical approach that focuses on probabilities. The metrics in the feature vector that are the input to the model are evaluated and based on

their correlation to existing labeled examples, the model predicts the probability of a power consumption reduction and decides to consolidate or not [32].

5.4 Training Data Generation

Training data was generated by evaluating single and multi-program workloads and collecting the four derived metrics: *shared line access rate, LLC miss rate, dTLB miss rate, and iL1 stall rate*. Single and two process workloads were launched with two threads, a native dataset, and all three of the available affinity configurations: OS-set, distributed and consolidated. Once all of the workloads' data had been collected, it was converted to a compatible format, suitable to use as input for the machine learning model.

Figure 8 shows a sample of the training data format. Columns are labeled for ease of understanding; however, the actual training files consist of only data. Each line in a training file consists of five values: four metrics, and a consolidation decision binary value. Four separate training files were generated for the single and multi-program workloads, each run with the default OS-set affinity, and a distributed affinity. The fifth column holds a binary value denoting whether that particular example is a case for consolidation and is determined by comparing execution times and looking for speedups. For example, the training file for single program execution run with an OS-set affinity (aff = 0), will compare the execution time of the workload run with the OS-set affinity and with a consolidated affinity. If there is a speedup with consolidation, that value is set to 1, otherwise it is set to 0.

Feature Vector < sharedAccess dTLB LLC iL1 >				**Consolidation** 0 = NO 1 = YES
SharedAccess	dTLB	LLC	iL1	
0.820311645	0.434068786	0.04371633	0.40430747	1
0.175179056	0.01092863	0.011303215	0.101745217	1
0.029067144	6.541132194	0.58791529	0.128706938	0
0.553497186	0.391143794	0.341219514	0.757300611	1
0.040902584	0.472901752	0.618969411	0.305489244	1
0.037385538	2.140303553	0.05255579	0.128473717	1
0.080016408	1.699219345	0.170007759	0.189180191	0
0.459672186	0.024041645	0.000992917	0.134337065	1
0.604480134	1.149189552	0.87351727	0.594297098	1
0.065993224	7.349141316	4.992606588	1.028080824	1
0.011935618	1.559747081	3.500501211	0.928137982	1
0.373058627	0.724450968	0.096821556	0.383440336	0

Figure 8: Sample training data

CHAPTER 6

Experimental Setup, Results and Evaluation

This chapter explains the experimental environment and presents the results of experiments for evaluating the various models. The first section provides details of the three platforms that workloads were run on followed by a description of the benchmark suite employed, and finally a thorough review of each workload. The chapter ends with a presentation of results and analysis, supplemented with comparative tables and graphs.

6.1 Experimental Setup

The workloads were executed and evaluated on three different systems to test the effectiveness and scalability of the migration policies, as well as to conform to the available performance counters provided by each of the systems. An overview of each system is presented followed by an inspection of the benchmark suite that is employed. Last of all a detailed description of workloads and their characteristics, concludes the experimental setup explanation.

6.1.1 Platforms

Each of the three utilized systems run Ubuntu Linux. The load balancing model was tested on a system with an Intel Sandy Bridge architecture. We refer to this platform as *Sandy Bridge I* in the remainder of this section. This system features a total of twelve cores, six physical cores which are hyper-threaded, and three levels of cache. Each core has a private L1 and L2 cache, and there is a shared L3 cache. There are a total of six cohorts where the cores are grouped as follows: (0,6), (1,7), (2,8), (3,9), (4,10), (5,11). The load balancing analytical model was the only model evaluated on this system for the

purposes of exploring model scalability, and another performance counter probing utility, 'likwid'.

The machine learning models were tested on a Sandy Bridge platform, featuring a Xeon processor with 2GHz clock rate, 16 physical cores and 32 logical cores. For the purposes of this research, 4 logical cores were utilized. The memory hierarchy of the system features 32KB L1 caches that are shared between two logical cores, 256 KB L2 caches also shared between two logical cores, and a 20MB L3 cache shared by all of the cores. We refer to this platform as *Sandy Bridge II* in the remainder of this section.

The resource sharing and machine learning models were both tested on a Core 2 Quad Core Intel Nehalem processor, which we will refer to as *Nehalem* for the remainder of this section. This platform features four processing cores, each with a private L1 cache, and two L2 caches, which are shared by cores 0 and 2, and cores 1 and 3 creating two cohorts or core groupings respectively. For the purposes of this study, these architectural features are the main focus.

6.1.2 Benchmarks

For all experiments, we use the Princeton Application Repository for Shared-Memory Computers (PARSEC) [33]. PARSEC consists of a set of multithreaded programs from various domains including computer vision, video encoding, financial analytics, animation physics, and image processing. The programs are examples of emerging types of workloads and give insight into the programming possibilities for maximizing parallelism. Table 2 lists some of the key characteristics of the PARSEC applications.

Table 2: PARSEC Benchmark suite program characteristics

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
x264	Media Processing	pipeline	coarse	medium	high	high

6.1.3 Workloads

Various workloads were generated to test each of the models. The programs are taken from the PARSEC benchmark suite, and are combined into different workloads designed to exhibit particular characteristics.

Both analytical models were tested using single and multi-program workloads, each with two, four and eight threads, and initiated with varying affinities. The load balancing analytical model was additionally tested with four program workloads to accommodate an increase in available cores when run on the *Sandy Bridge* platforms. Table 3 shows the programs included in the multi-program workloads used with the resource sharing model followed by Table 4 listing the four program workloads.

Table 3: Two-program workload characteristics

Workload A	Workload B	Workload C	Workload D	Workload E	Workload F
[1] blackscholes	[2] ferret	[6] raytrace	[0] bodytrack	[4] fluidanimate	[4] fluidanimate
[7] swaptions	[6] raytrace	[10] streamcluster	[7] swaptions	[5] freqmine	[7] swaptions

Workload G	Workload H	Workload I	Workload J	Workload K	Workload L
[5] freqmine	[2] ferret	[7] swaptions	[1] blackscholes	[4] fluidanimate	[6] raytrace
[8] x264	[7] swaptions	[8] x264	[8] x264	[10] streamcluster	[6] raytrace

Table 4: Four-program workload characteristics

Workload A	Workload B	Workload C	Workload D	Workload E	Workload F
[0] bodytrack	[0] bodytrack	[1] blackscholes	[0] bodytrack	[1] blackscholes	[9] canneal
[3] facesim	[2] ferret	[7] swaptions	[5] freqmine	[1] blackscholes	[9] canneal
[7] swaptions	[4] fluidanimate	[9] canneal	[7] swaptions	[10] streamcluster	[10] streamcluster
[7] swaptions	[5] freqmine	[9] canneal	[8] x264	[10] streamcluster	[10] streamcluster

Three initial affinity configurations were explored when first launching a workload. The first of the three allows the OS to decide the affinity of the workload and reflects the decision of the default scheduler (aff = 0). The second affinity option is a worst-case scenario which assigns tasks in a configuration that maximizes physical distance between tasks, consequently increasing the amount of memory access in the event of shared resources (aff = 2). The last affinity configuration promotes consolidation and assigns tasks to core groups or cohorts, ensuring a certain amount of available shared cache resources (aff = 5).

The last tunable parameter available when launching a workload is the number of threads that each program in the workload will be executed with. Different thread counts were evaluated to pinpoint differences in trials where the thread count was less than,

equal to, and greater than the number of available cores. Exploring different thread counts allowed for closer analysis and a better simulation of real world applications.

The training data used by the machine learning model was generated by running a number of single and multi-program workloads. Each workload was executed with two threads and with three different initial affinities. Evaluating the two-program workloads described earlier generated additional training data.

Table 5 shows a comprehensive listing of the workloads evaluated for each of the proposed models, and the metrics used for final analysis.

Table 5: Workloads evaluated with each migration model

Load Balancing Analytical Model			
Workloads	Affinity	Threads	Objective Metric
multi-program	0, 2	4, 8, 16	execution time, power
Resource Sharing Analytical Model			
Workloads	Affinity	Threads	Objective Metric
single program	0, 2, 5	2, 4, 8	execution time
multi-program	0, 2, 5	2	execution time
Machine Learning Model			
Workloads	Affinity	Threads	Objective Metric
single program	0, 2, 5	2	execution time, power
multi-program	0, 2, 5	2	execution time, power

6.2 Power Reduction with Load Balancing

The load balancing model was tested on the *Sandy Bridge I* with a number of different workloads and varying parameters. Shown below are the power consumption levels of eight different four-program workloads, where each program is launched with four and eight threads. Each workload was run twice; once with the affinity set to default OS scheduler, and once with an initial distributed affinity and the load balance model

handling the final affinity configuration. Figure 9 below shows the power consumption for a large dataset, followed by Figure 10 featuring power consumption levels for a native dataset, both for four threaded applications.

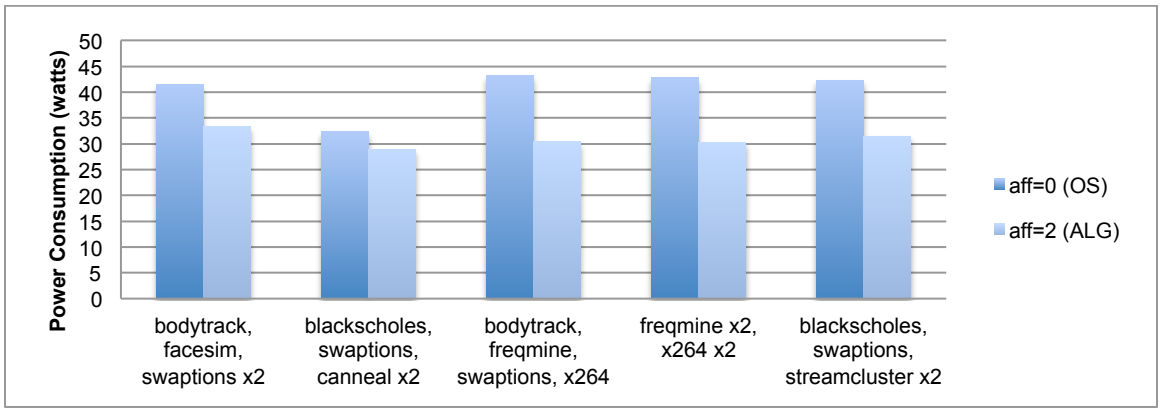

Figure 9: Power consumption- 4 program workload, 4 threads, large dataset

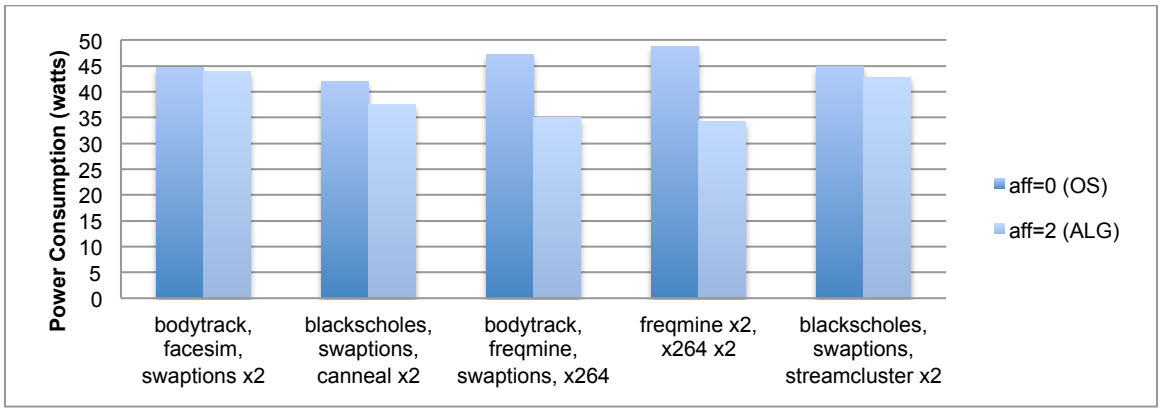

Figure 10: Power consumption- 4 program workload, 4 threads, native dataset

In 100% of the experiments, the resource sharing model provides a thread migration solution that consumes less power than the traditional OS scheduler.

Figures 11 and 12 follow with power consumption levels of the same workloads launched with eight threads per application.

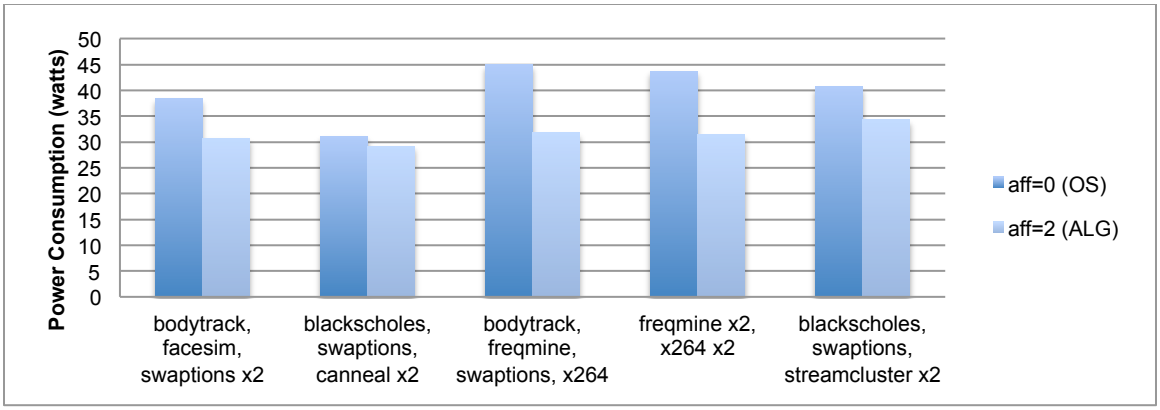

Figure 11: Power consumption- 4 program workload, 8 threads, large dataset

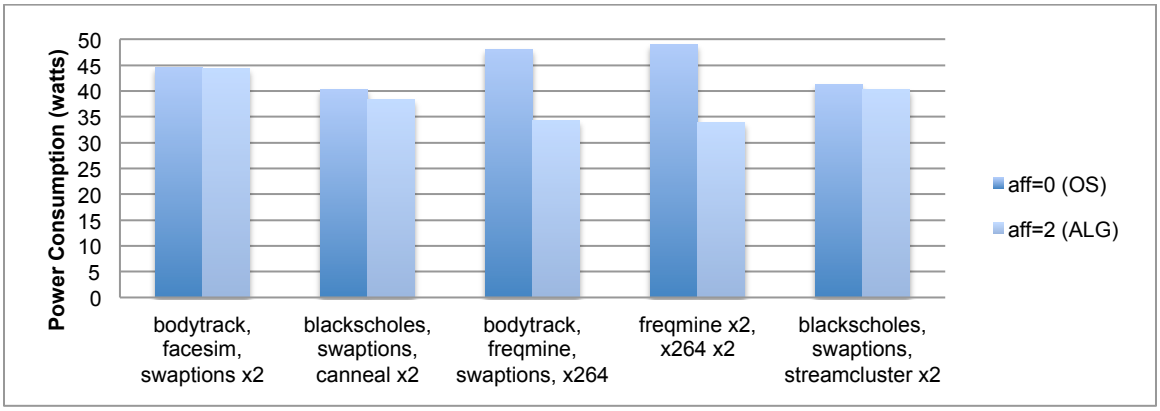

Figure 12: Power consumption- 4 program workload, 8 threads, native dataset

We see the same power efficiency improvement with the resource sharing model as the working dataset size is significantly increased, effectively developing a strategy to operate within an established power cap. Figure 13 provides a summary of the average power savings in percent that the load balance model provides, per workload, for the experiments shown in Figures 9-12.

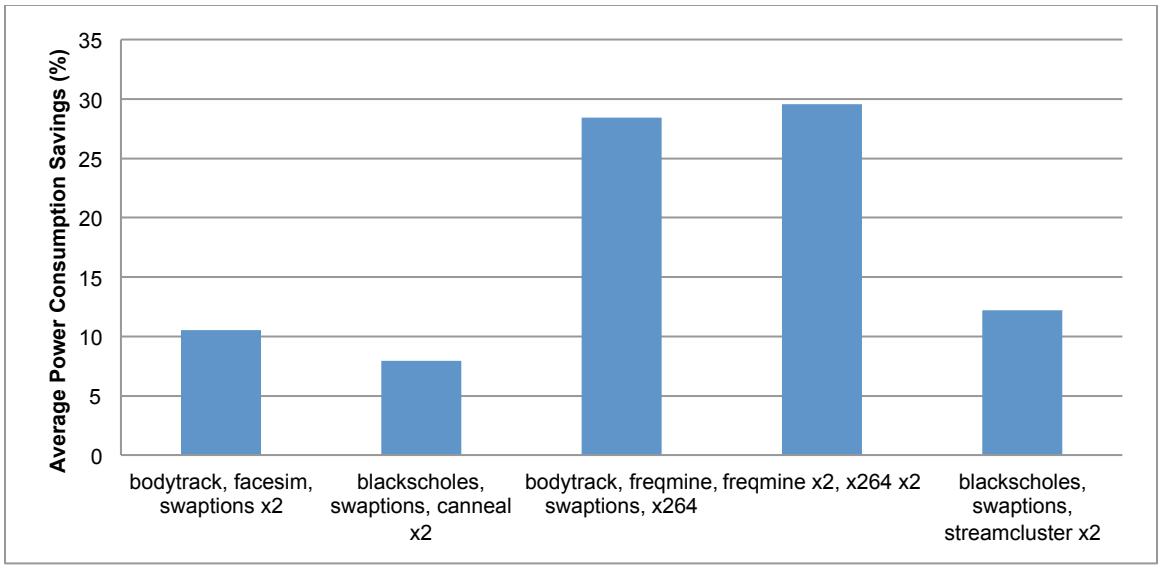

Figure 13: Average power consumption savings in percent by the load balancing model per workload, *Sandy Bridge I* platform

Figure 13 shows that in three out of the five workloads, we see that the load balancing model provides power consumption savings of ~10%. We see a significant increase in savings for two of the workloads, both seeing greater than a 25% decrease in power consumption.

6.2.1 Overhead

The overhead of the load balancing model was calculated by running the evaluated workloads with certain parameters. The execution times of the four program workloads run with large and native datasets, four threads, and with OS-set affinities are first collected. Next, the same workloads are run through the load balancing model script, and a flag is set so that all of the work by the script is completed, except for the actual migrations, in which case the 'taskset' command is not executed. This allows for an evaluation of the model's overhead in terms of execution time. Figures 14 and 15 compare the execution times for each workload run with and without the load balancing model invoked for large and native datasets, establishing the overhead.

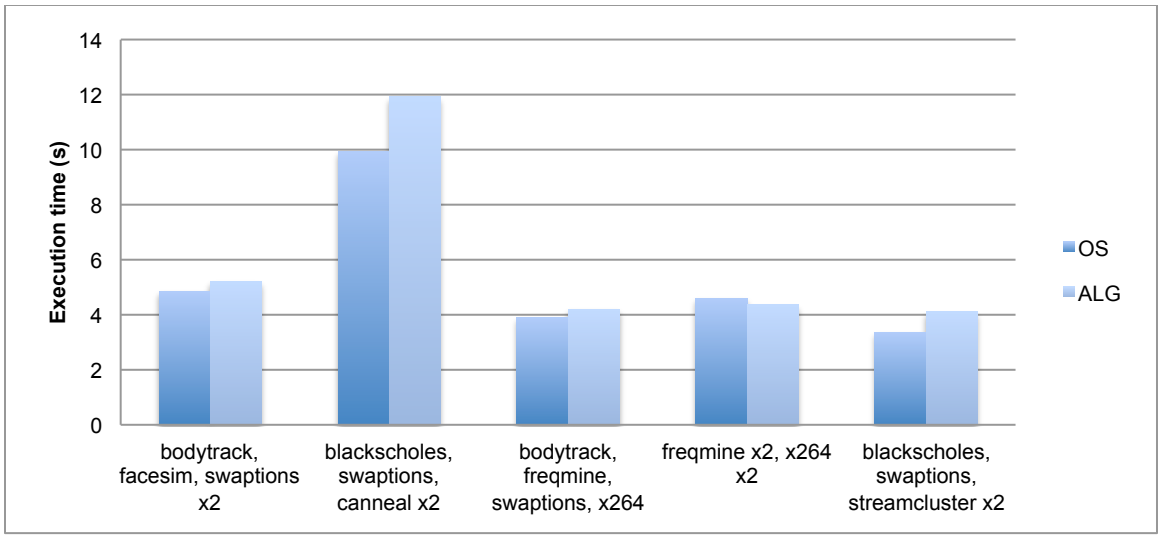

Figure 14: Load balance model overhead- 4 program workload, 4 threads, large dataset

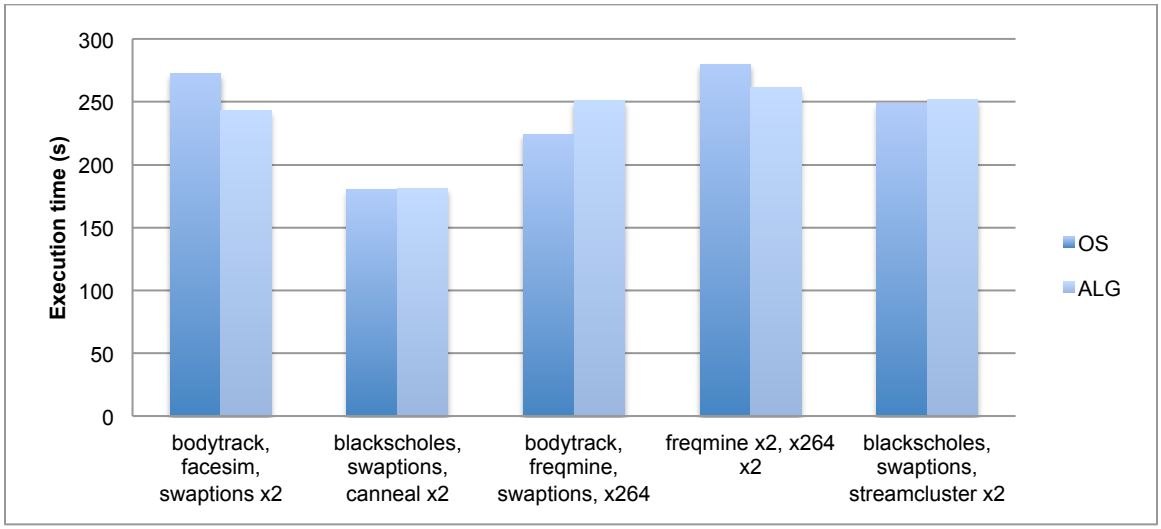

Figure 15: Load balance model overhead- 4 program workload, 4 threads, native dataset

6.2.2 Summary

The goal was to establish that there is a power cap that most systems must run under, and the model successfully and consistently provides a migration policy that consumes less power than the default scheduler. In all cases shown, regardless of what any specific workload characteristics are, the load balancing model provides an affinity

configuration that balances out the load of the system and consumes less power. Additionally, the model is able to overcome an initial distributed affinity, and still provide that power savings. Due to a performance tradeoff, however, additional considerations must be taken in order to provide a more efficient scheduler.

6.3 Workload Speed Up with Resource Sharing

The resource sharing model was evaluated by comparing the execution times of a workload launched with varying parameters. Two types of workloads were explored: single program and multi-program. Each workload was launched with two, four, and eight threads, and its execution time was recorded to reflect the speedup of the resource sharing model over the OS set affinity, and a distributed affinity. The model was found to be the most effective when each of the workloads was executed using two threads and is reported here. For single program, two threaded workloads, and a native dataset, Figure 16 illustrates the overall speedup of the resource sharing migration model over the default OS scheduler. Figure 17 follows with the overall speedup of the migration model over a distributed affinity.

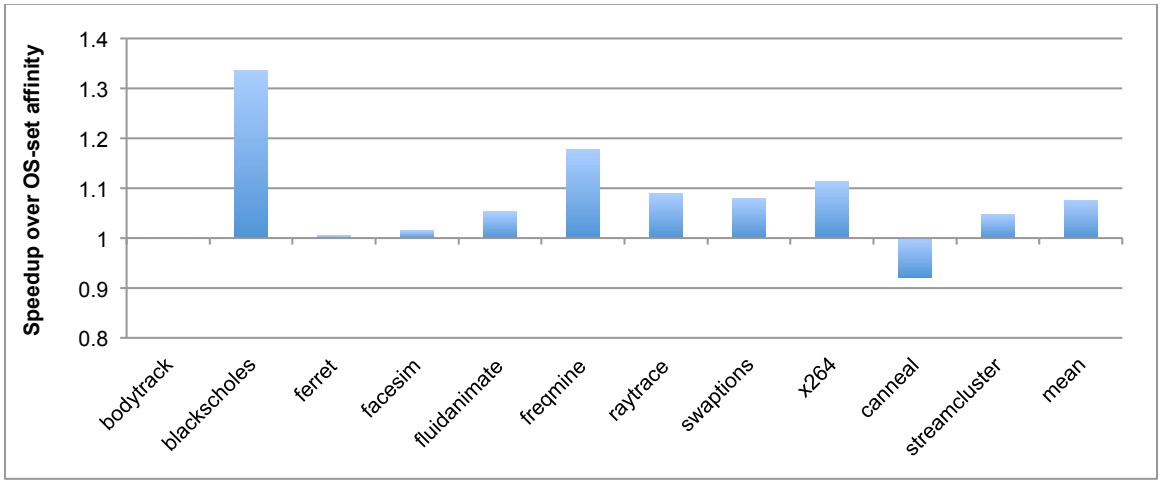

Figure 16: Speed up of the resource sharing model over OS-set affinity, single program workload, 2 threads, native dataset

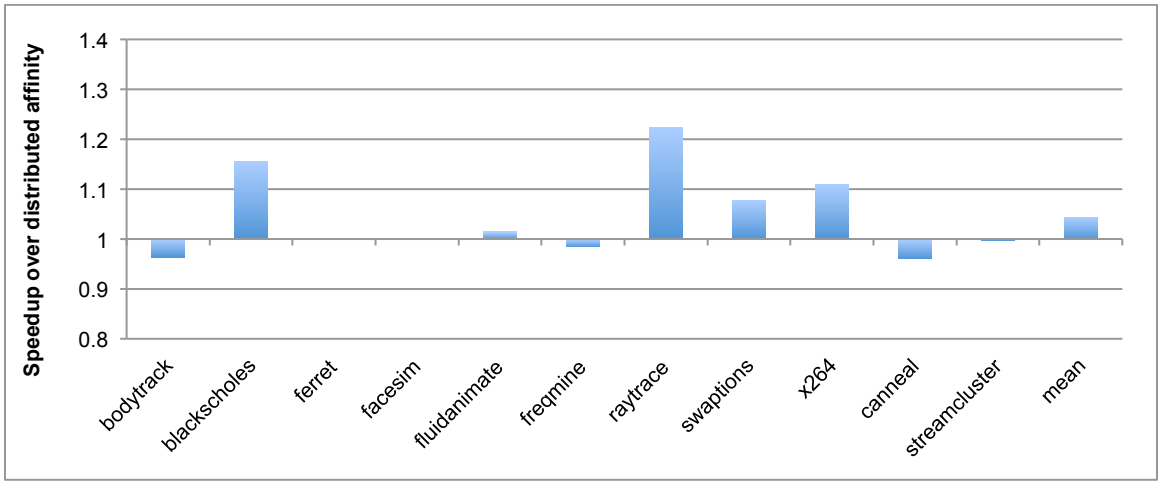

Figure 17: Speed up of the resource sharing model over a distributed affinity, single program workload, 2 threads, native dataset

The single program workloads help to identify some interesting results. The greatest speedup when employing the resource sharing model over both the OS-set affinity and a distributed affinity, occurred when running the following applications: blackscholes, raytrace, swaptions and x264. These programs exhibit characteristics spanning opposite ends of the ranges. For example, blackscholes features a small working set and low data sharing. Raytrace, on the other hand, utilizes an unbounded working set and features high data sharing. Swaptions and x264 both use a medium size working set, but require low and high data sharing respectively. The model makes a successful decision to either consolidate or not in the above cases, regardless of characteristics, that results in improved execution times, and consequently reduces power consumption.

For multi-program workloads, the data is represented in the same manner as with single workloads. Figure 18 shows the speed up of the resource model over the default

scheduler, and Figure 19 illustrates the speed up over a distributed affinity.

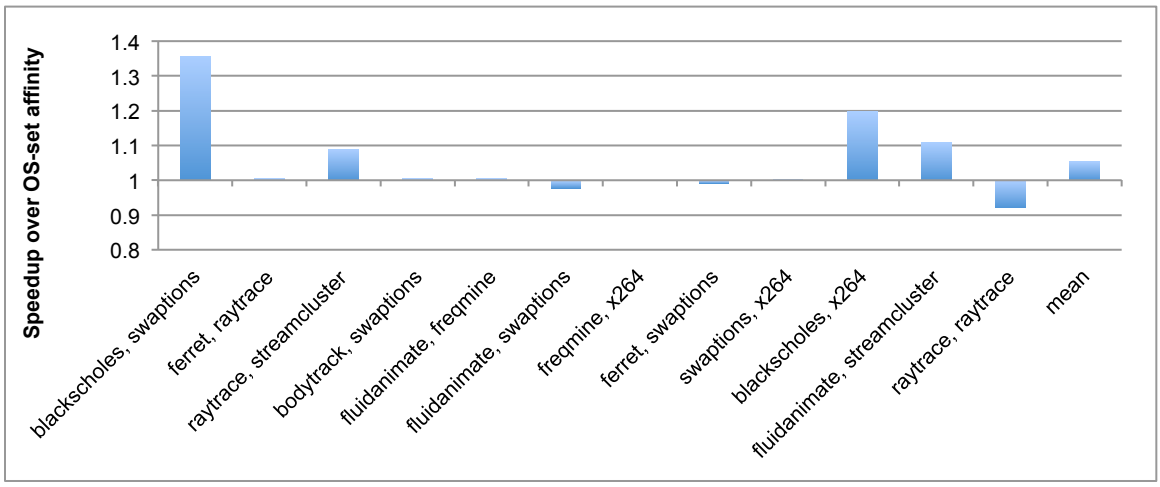

Figure 18: Speed up of the resource sharing model over OS-set affinity

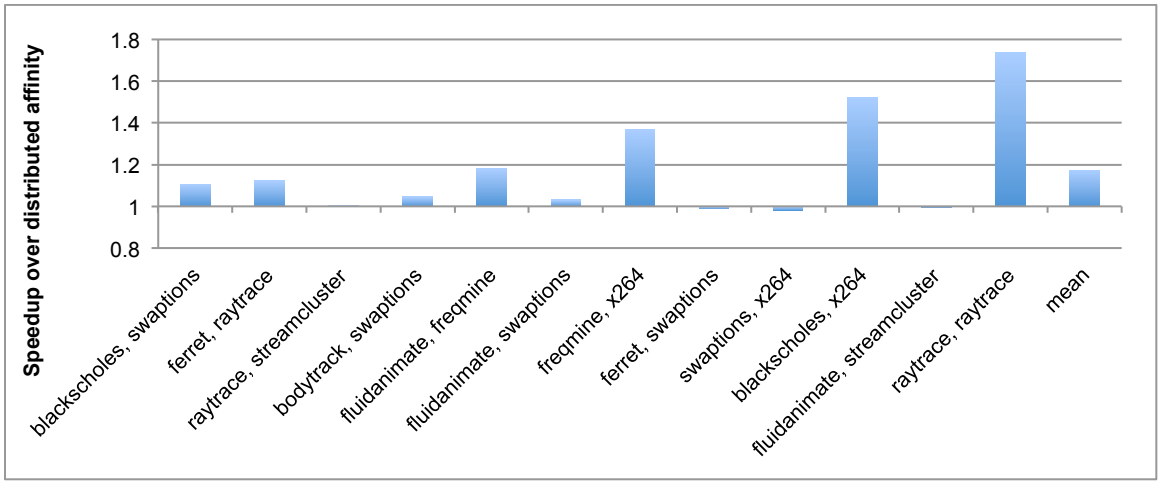

Figure 19: Speed up of the resource sharing model over a distributed affinity

The multi-program workloads offer additional insight into the behavior of the model. The greatest speed up over an OS-set affinity is seen with the following multi-program workloads: blackscholes and swaptions, and blackscholes and x264. Here a combination of small and medium working sets are found, as well as data sharing ranging from low to high. This implies that the model is capable of making consolidation

decisions based on those shared resources regardless of the degree of sharing. The greatest speed up over a distributed affinity is again seen with blackscholes and x264, as well as with two instances of the raytrace application. Raytrace features an unbounded working set and high sharing. Launching two instances of raytrace with a distributed affinity would force half of the threads to access memory not in their respective cohort. The resource sharing model essentially consolidates one instance of raytrace to one cohort, and the other instance to the other. The result is a significant increase in runtime, thereby effectively reducing power consumption.

6.3.1 Overhead

A final comparison in execution times between a workload executed with a consolidated affinity using both the default scheduler and the resource sharing model reveals the amount of overhead that the model incurs. Figures 20 and 21 show the overhead of the model by comparing the runtime of the same workloads started with a consolidated affinity, and scheduled with the default scheduler and with the resource sharing model.

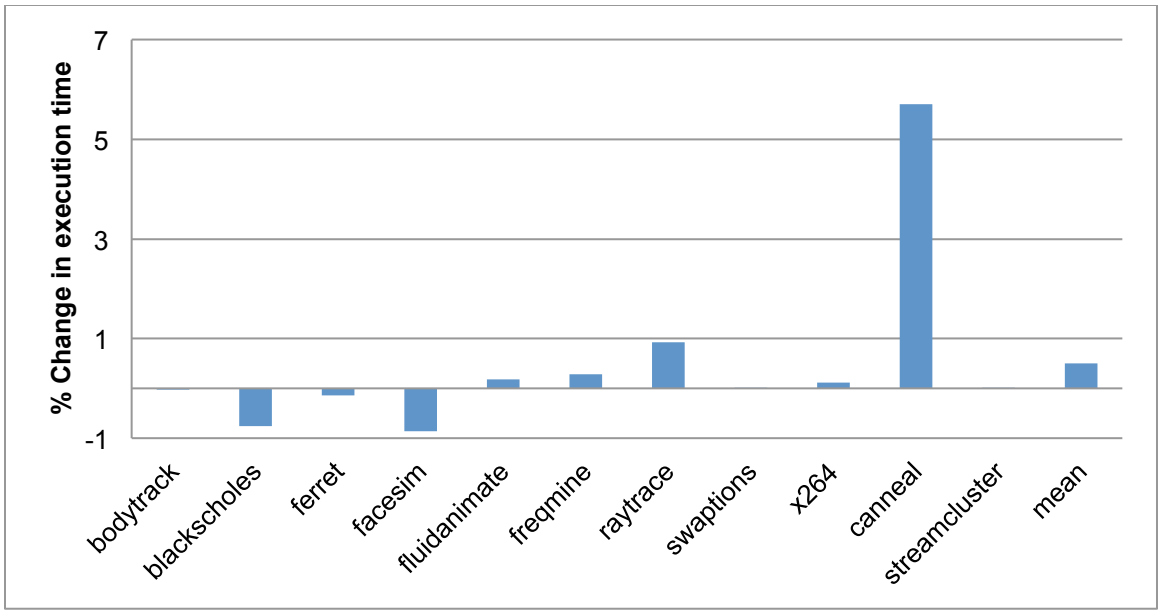

Figure 20: Overhead of the resource sharing model for single program workloads

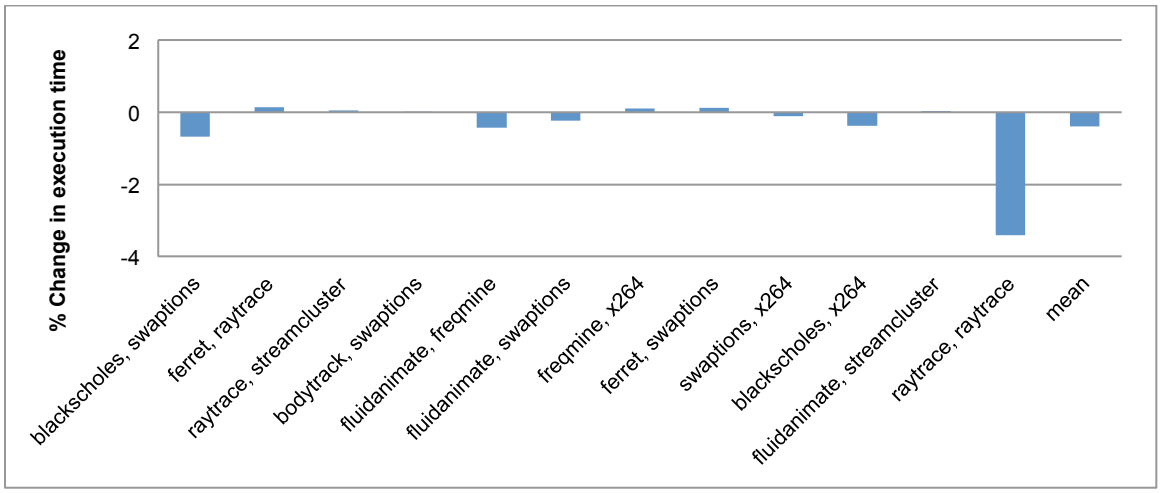

Figure 21: Overhead of the resource sharing model for multi-program workloads

6.3.2 Summary

The resource sharing model increases the amount of utilized dynamic feedback and considers a total of four performance metrics. This yields a more judicious model than what the previous greedy algorithm provided, and reduces the performance trade-off seen with the load balancing migration policy. The results presented reflect single

program and multi-program workloads. Table 6 shows a summary of the resource model's overall speedup over an OS-set affinity, and a distributed affinity.

Table 6: Percentage of experiments where the resource sharing model provided a speed up in execution time

	% of experiments where the model provided speed up over OS-set affinity	% of experiments where the model provided speed up over distributed affinity
Single Program	83.33%	50.00%
Multi-program	66.67%	75.00%

From Table 6, it is evident that reducing shared resource contention between concurrently running programs reduces execution time a majority of the time.

6.4 Energy Efficiency with Machine Learning Models

The three machine learning models were tested on two systems: *Nehalem* and *Sandy Bridge II*. Single workloads were first run with large and native datasets, and execution times are compared between the default OS scheduler handling the migration and the three algorithms: decision tree, SVM, and Bayes. Figure 22 presents the execution times of the OS scheduler and the SVM model for a large dataset.

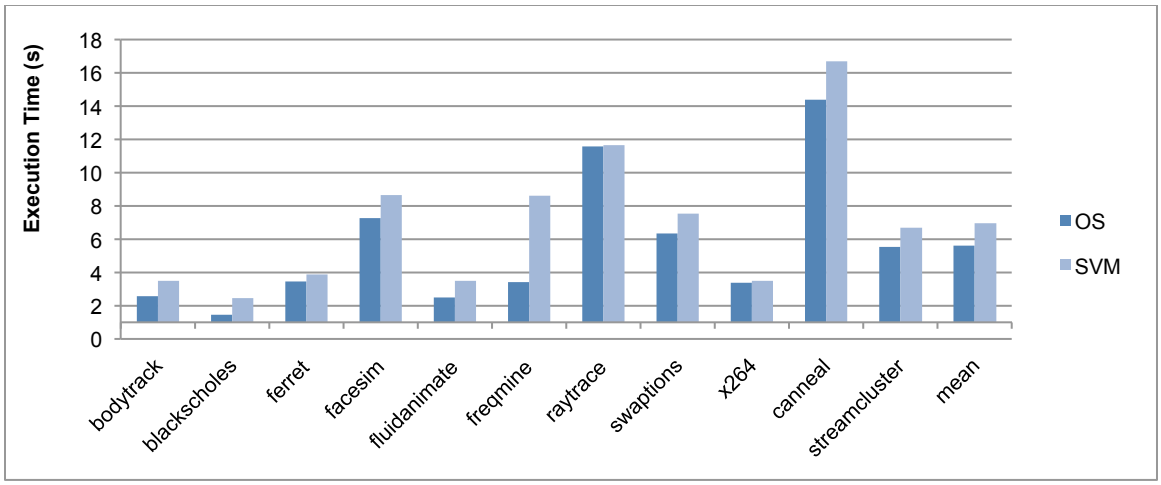

Figure 22: OS scheduler and SVM ML model execution times- single workload, 2 threads, large dataset, *Nehalem* platform

Similar results were found for the other two algorithms, so attention is directed to multi-program workloads. Figures 23-25 show the execution times for two program workloads launched with two threads each, and a large dataset. A comparison is made between the times when the OS schedules and when each of the models handles the scheduling.

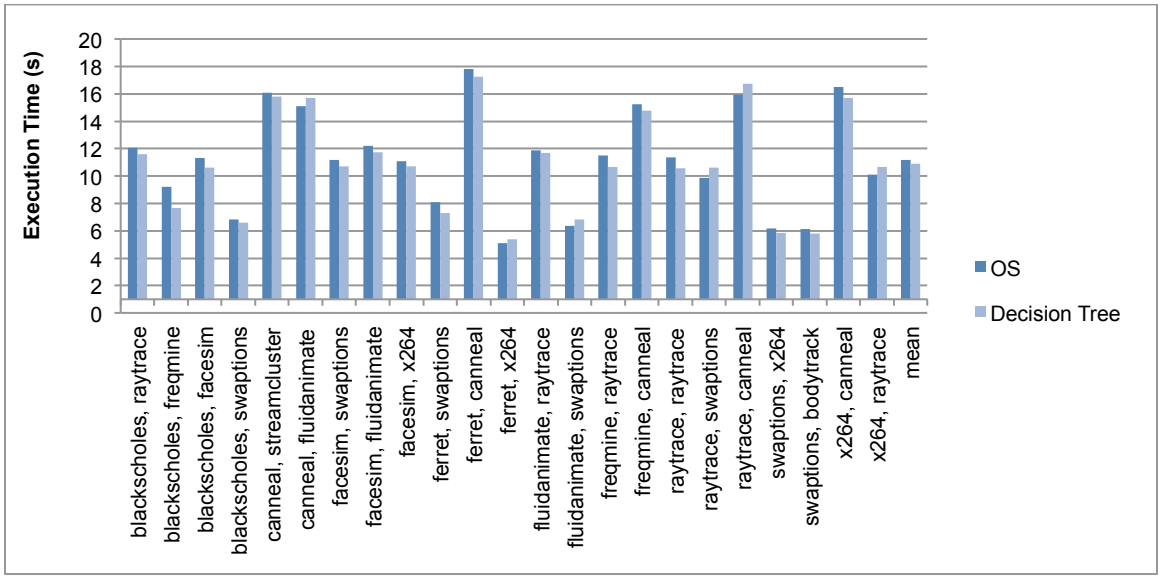

Figure 23: OS scheduler and Decision Tree ML model execution times- multi-program workload, 2 threads, large dataset, *Nehalem* platform

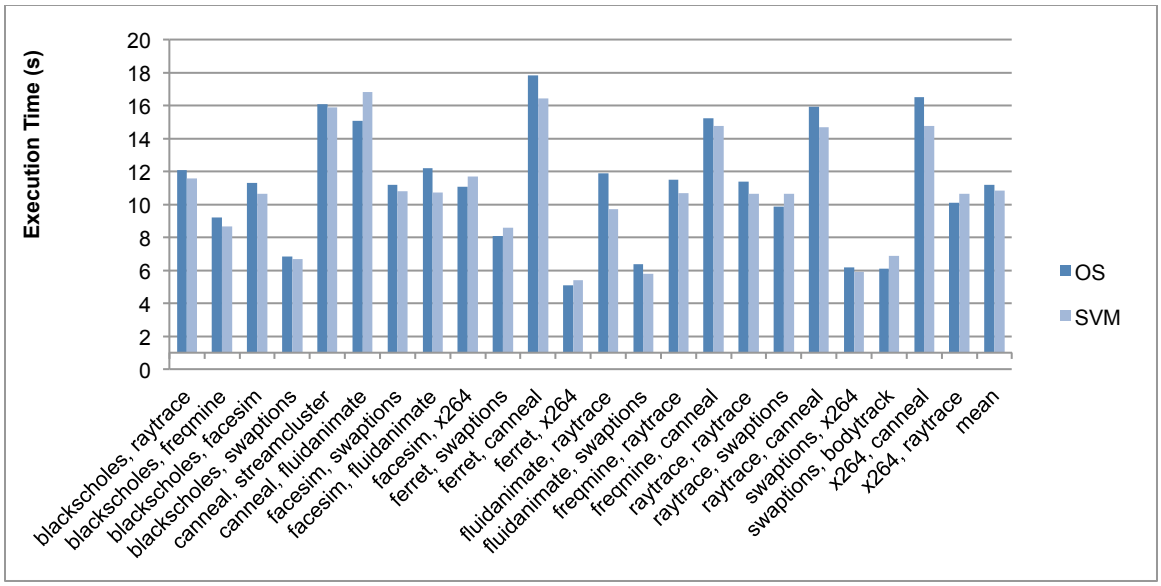

Figure 24: OS scheduler and SVM ML model execution times- multi-program workload, 2 threads, large dataset, *Nehalem* platform

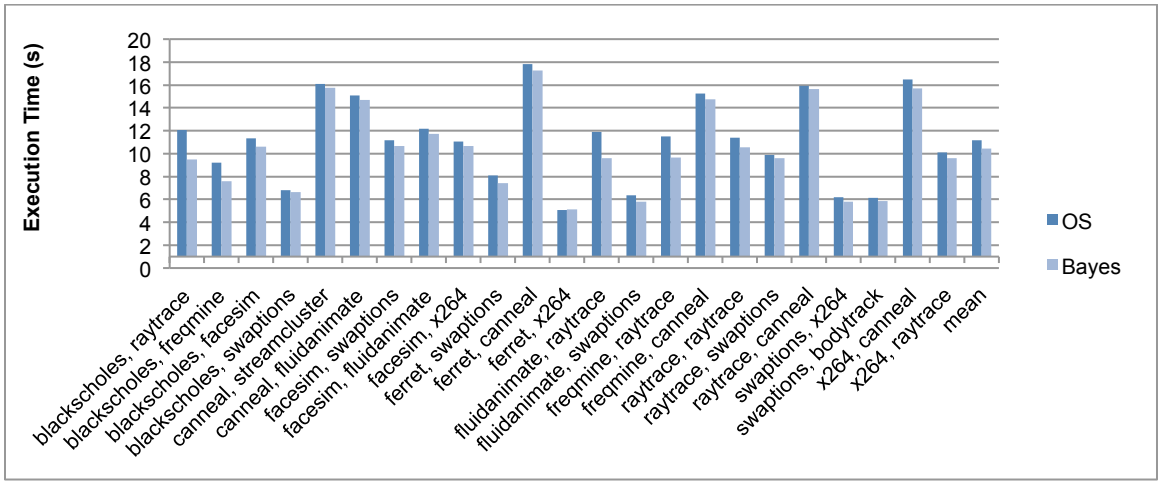

Figure 25: OS scheduler and Bayes ML model execution times- multi-program workload, 2 threads, large dataset, *Nehalem* platform

The execution times of all three models in comparison to the OS scheduler, on average, were consistently less for multi-program workloads using large datasets.

Single program workloads with native datasets were also run to obtain additional insight to possible behavior differences that may be data dependent. Figure 26 features the execution time comparison between the OS scheduler and a decision tree machine

learning model. Figure 27 shows the same comparison using a SVM model, and Figure 28 features the Bayesian model.

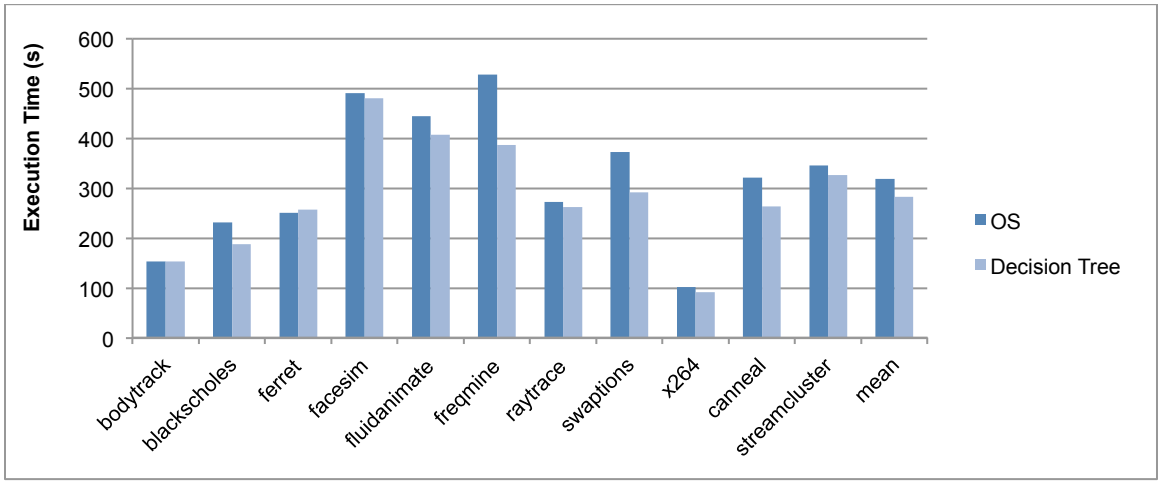

Figure 26: OS scheduler and Decision Tree ML model execution times- single workload, 2 threads, native dataset, *Nehalem* platform

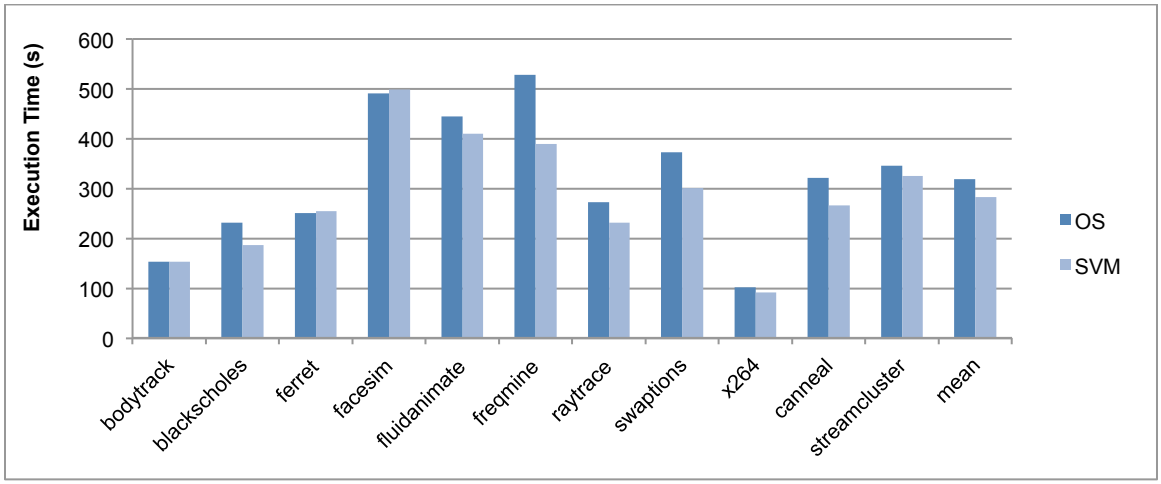

Figure 27: OS scheduler and SVM ML model execution times- single workload, 2 threads, native dataset, *Nehalem* platform

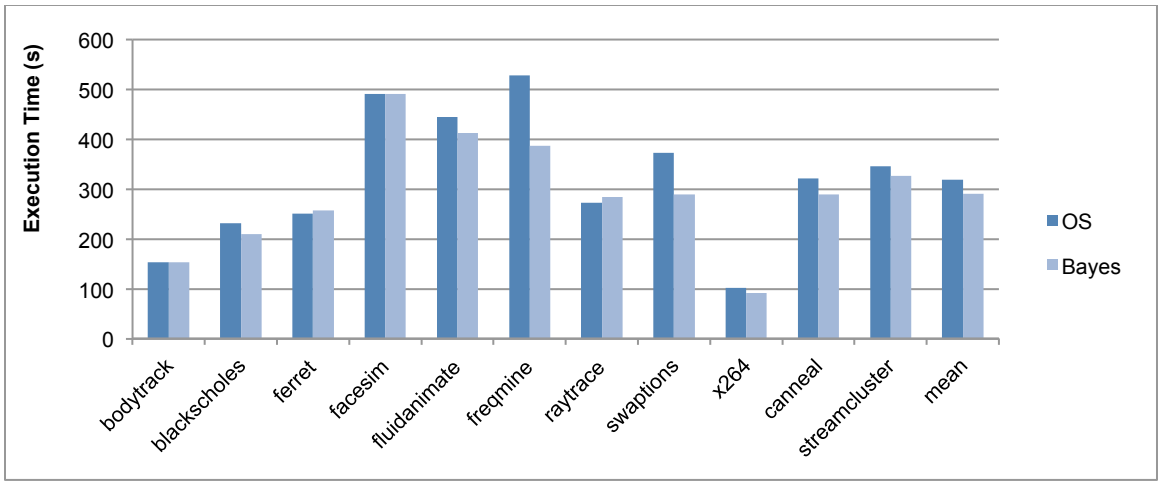

Figure 28: OS scheduler and Bayesian ML model execution times- single workload, 2 threads, native dataset, *Nehalem* platform

Again, we see the same consistent behavior from the machine learning models, with single program workloads and native datasets. Certain programs don't see the same time improvement from the machine learning models such as bodytrack, ferret and facesim, and all three programs vary in working set size and sharing. This implies that there may be thread activity within the program that inhibits the machine learning model to provide better performance and requires further investigation.

The machine learning model was also evaluated on the *Sandy Bridge II* platform. Figure 29 shows the highest obtainable gain in execution times, energy and power for single process workloads run with a native dataset and two threads, if the three machine learning models were to be combined.

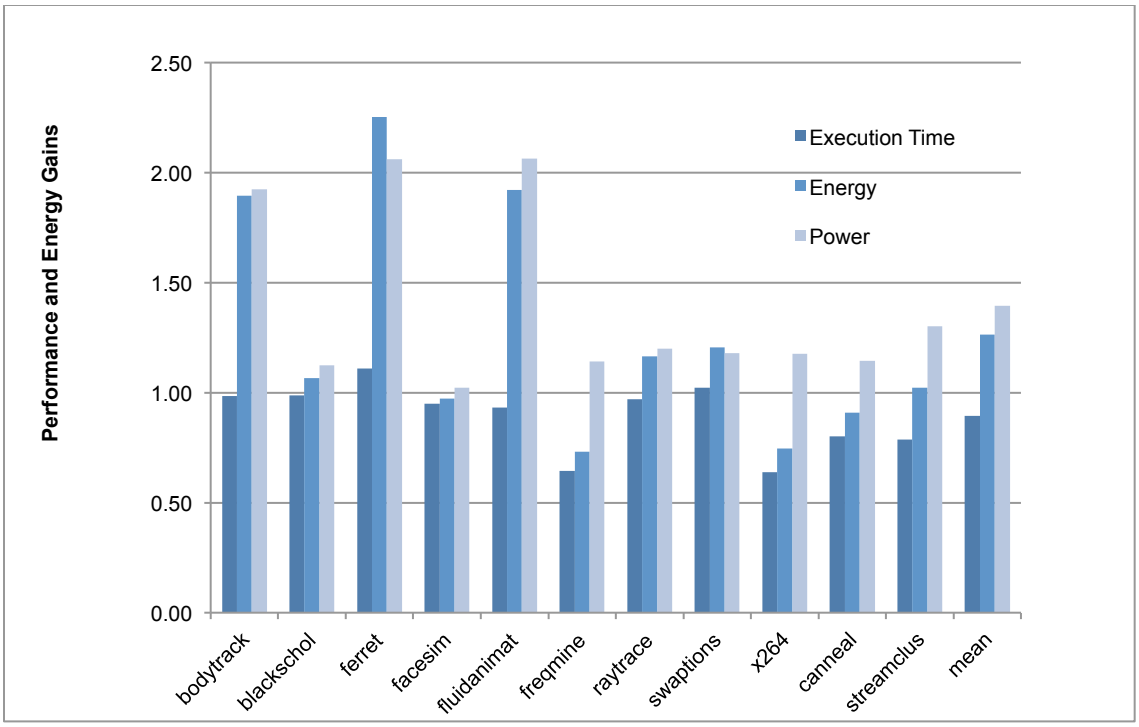

Figure 29: Highest obtainable gain by combining all three machine learning models

Figure 29 shows that the greatest gain offered by the machine learning model is power savings. On average, the model provides about 1.4 times less power consumption than the OS scheduler which can be significant in certain applications.

6.4.1 Model Accuracy

To evaluate the quality of the machine learning models, we performed a 10-fold cross validation during training. In this process, models were trained on 90% of the training data points and then evaluated on the remaining 10%. This process was repeated ten times, where each time, the 10% of the data points to be left out was selected randomly.

Table 7 presents the results of the cross-validation on the three models. We report the average percentage of correctly classified instances for both single-program and

multi-program workloads. 'Correctly classified' implies that the model was able to pick the best known thread configuration for the workload.

Table 7: Average percentage of correctly classified instances of workloads by machine learning models

	Single-Program	Multi-Program
Decision Tree	75%	100%
SVM	75%	83%
Bayes	79%	83%

We observe that the accuracy of all three models is lower for the single program workloads. This is expected, as the number of samples was much smaller in this case. Somewhat surprisingly, the accuracy does not improve markedly for SVM or Bayes when moving to multi-program workloads. We do see a significant improvement in Decision Tree, which yields almost perfect accuracy.

6.4.2 Summary

The single program workloads that were tested using a large dataset did not benefit as much from the machine learning model's scheduling policy. Due to the overhead of the model and the workloads completing within a few seconds, obtaining better performance was limited. An increase in the model's performance with a large dataset was seen when an additional program was added to the workload. The multi-program workloads run with the decision tree algorithm executed quicker than the OS scheduled workload 74% of the time. The SVM model performed better 70% of the time, and the Bayesian model had increased performance 96% of the time. Native datasets have much greater execution times, minimizing the effect that the model's overhead has on overall performance. The decision tree model provided a mean speedup

over the OS scheduler of 1.126 or roughly 12.6%. The SVM model provides an average speedup of 1.13 or 13%, and the Bayesian model shows a speedup of 1.13 or 13%. When combining all three models, the greatest possible expected gain is ~1.25.

CHAPTER 7

Conclusion

The demand for power efficient computing offers an opportunity for innovative methods to be adopted when considering task scheduling in processors. Though traditional OS schedulers do take several considerations when making decisions, power efficiency is often overlooked. This research presents a novel approach to developing a power aware migration policy capable of utilizing dynamic feedback to guide decisions. Three different approaches are evaluated including two analytical models and one machine learning method.

First, a greedy algorithm is developed with the goal of providing a migration policy that promotes a balanced load to the available processors, and can operate within an established power cap. A continuous evaluation of the workload's effect on each core's utilization level helps to determine the migration of individual threads. The 'greedy' nature of the algorithm yields a method that is successful at reducing power consumption with a performance trade-off.

The second proposed model considers shared resources. Additional hardware performance counters are probed to determine whether consolidation would be a power saving option. Once the decision is made, execution is resumed and allowed to finish.

The last approach is a machine learning approach that is trained on the same metrics that the shared resource model employs. The trained model evaluates certain performance counters at runtime, and decides if consolidation is beneficial or not. Once the decision is made, execution is allowed to resume. Three different algorithms are explored to offer various perspectives.

An evaluation of each model reveals certain strengths and weaknesses in approach. Various single and multi-program workloads are developed that exhibit known effects on available hardware resources. Every workload's performance is evaluated when scheduled with the default scheduler, and with each proposed model. Results indicate that an increase in metric consideration positively effects both runtime and power consumption, and that a machine learning approach can be a very effective technique for energy efficiency through thread migration.

7.1 Recommendations for Future Work and Research Expansion

The following proposals offer opportunity for better performance and possibly greater power savings.

1. A new model can be developed that combines the metrics employed with the load balancing and shared resource models. Factoring in both core utilization and shared resources may offer additional benefits due to the added consideration.
2. Rather than evaluating the combined effect of the three algorithms employed with the machine learning approach, a heuristic can be developed that successfully opts to use the algorithm with the best expected performance given a particular workload's characteristics.
3. Workloads can be modified to include a greater amount of parameter variation, introducing new scenarios for evaluation and training data collection.

REFERENCES

[1] *The Linux Kernel Archives.* The Linux Kernel Association. Web. 19 June 2014.

[2] R. Love. (2005) *Linux Kernel Development 2nd Ed.* Retrieved from http://www.makelinux.net/books/lkd2/ch04lev1sec2

[3] S. Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness, MSPC '08, ACM,* pages 26-30, 2008.

[4] S. Sarangkar.and A. Qasem. A Model-driven Adaptive Tuning System for Parallel Workloads. In *Journal of Parallel and Cloud Computing*, pages 50-64, 2012.

[5] S. Rahman and R. Hay. Enhancing Learning-based Autotuning with Composite and Diagnostic Feature Vectors. In *26th International Conference on High Performance Computing, Networking, Storage and Analysis.* 2013.

[6] M. Burtscher, B. Kim, J. Diamond, J. McCalpin, L. Koesterke and J. Browne. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* 2010.

[7] M. Tikir and J.K. Hollingsworth. Using Hardware Counters to Automatically Improve Memory Performance. In *Proceedings of the ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis.* 2004.

[8] J. Chen, L.K. John, and D. Kaseridis. Modeling program resource demand using inherent program characteristics. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (San Jose, CA, 2011) SIGMETRICS '11, ACM,* pages 1-12, 2011.

[9] K. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *Proceedings of Parallel and Distributed Processing Symposium(2005).*

[10] G. Contreras, and M. Martonosi. Power prediction for Intel XScale reg; processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pages 221-226, 2005.

[11] K. Singh, M. Bhadhauria, and S. McKee. Real time power estimation and thread scheduling via performance counters. In *ACM SIGARCH Computer Architecture News*, pages 46–55, May 2008.

[12] R. Azimi, D.K. Tam, L. Soares and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. In *SIGOPS Oper. Syst. Rev.(New York, NY, 2009), ACM,* pages 56-65, April 2009.

[13] Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 290–301, 2009.

[14] O. Sarood, and L. V. Kale. A 'Cool' Load Balancer for Parallel Applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2011), SC '11, ACM*, pages 21:1–21:11.

[15] E. Musoll. Energy and thermal tradeoffs in hardware-based load balancing for clustered multi-core architectures implementing power gating. In *Proceedings of the 2008 IEEE Symposium on Application Specific Processors, SASP '08*, pages 89 –94, 2008.

[16] Z. L. Zong, A. Manzanares, X. J. Ruan, and X. Qin. EAD and PEBD: Two Energy-Aware Duplication Scheduling Algorithms for Parallel Tasks on Homogeneous Clusters. In *IEEE Transactions on Computers,* Vol. 60, Issue 3, pages 360-374, March 2011.

[17] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08,* 2008.

[18] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07,* pages 47–58, 2007.

[19] J.A. Brown, L. Porter, and D.M. Tullsen. Fast thread migration via cache working set prediction. In *Proceedings of HPCA'11*, Feb 2011.

[20] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhang. Cache topology aware computation mapping for multicores. In *PLDI '10*, pages 74–85, New York, NY, USA, 2010. ACM.

[21] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA'07*, pages 105–115, 2007.

[22] A. Merkel, J. Stoess and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer system, EuroSys '10*, 2010.

[23] S. Boyd-Wickizer, R. Morris and M.F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th conference on Hot topics in operating systems, HotOS '09*, 2009.

[24] D. Tam, R. Azimi and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[25] M. Banikazemi, D. Poff and B. Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[26] Vega, P. Bose, A. Buyuktosunoglu. Power-Aware Thread Placement. In *SMT/CMP Architectures*. Workshop on Energy Efficient Design (WEED), in conjunction with ISCA, June 2012.

[27] Vega, P. Bose, A. Buyuktosunoglu. SMT-centric power-aware thread placement in chip multiprocessors. In *PACT 2013 Proceedings of the 22nd international conference on Parallel architectures and the compilation techniques*, pages 167-176.

[28] Decision tree learning. (n.d.). In *Wikipedia.* Retrieved October 21, 2014, from http://en.wikipedia.org/wiki/Decision_tree_learning

[29] Support vector machine. (n.d.). In *Wikipedia.* Retrieved October 21, 2014, from http://en.wikipedia.org/wiki/Support_vector_machine

[30] Bayesian network. (n.d.). In *Wikipedia.* Retrieved October 21, 2014, from http://en.wikipedia.org/wiki/Bayesian_network

[31] J. Treibig, G. Hager, G. Wellein, and M. Meier. 2011. Poster: LIKWID: lightweight performance tools. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion (SC '11 Companion)*. ACM, New York, NY, USA, pages 29-30. http://doi.acm.org/10.1145/2148600.2148616

[32] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N.R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[33] *Parsec- A unit of measure.* Princeton University. 2007-2010. Retrieved July, 2013, from http://parsec.cs.princeton.edu