Energy-Efficient Thread Migration via Dynamic Characterization of Resource Utilization

Claudia Alvarado

Intel Corporation Portland, OR
ca1015@txstate.edu

Dan Tamir and Apan Qasem

Department of Computer Science
Texas State University, San Marcos, TX
{dt19, apan}@txstate.edu

***Abstract*—The affinity of threads with cores in current chip-multiprocessor systems has a substantial impact on the execution time, latency, and power consumption of multi-threaded workloads. Finding an optimal mapping configuration of threads is a significant challenge as it requires detailed knowledge of each thread's demands for shared system resources. This paper describes a software-based strategy that makes judicious thread migration decisions founded on careful inspection of dynamic resource utilization. The main novelty of the system reported in this paper is the extensive utilization of hardware performance counters to develop a set of synthesized metrics that capture resource contention among co-running threads. Experimental results with a set of contemporary parallel workloads, show that the system can achieve significant improvements in power consumption and performance over the default scheduling heuristics implemented in the Linux kernel.**

Keywords–energy efficiency; thread scheduling; workload characterization.

I. INTRODUCTION

The proliferation of portable wireless devices as well as the rapid growth of high-performance server farms and data centers have made power consumption a central point of concern for the entire computing industry. Excessive and unbalanced power consumption of computing systems has a direct economic and environmental impact in the form of high energy bills and large carbon footprints. Additionally, power consumption impacts the computing industry in several indirect ways by adverse effect on device reliability, requiring expensive packaging, and causing irreversible damage to semi-conductor devices. As the industry moves towards the exascale era and the magnitude and volume of computing devices continue to grow, it is clear that power consumption is a dominant metric in the design of computing systems. Several strategies for power-reduction have emerged in response to this challenge in recent years. Several, researchers have focused on hardware techniques such as developing new energy conserving components while others have emphasized software strategies that aim to exploit existing hardware in an energy-efficient manner. This paper focuses on software strategies addressing the problem of determining suitable thread placement policies that improve the energy efficiency of multi-threaded workloads without a commensurate sacrifice in performance.

Power consumption on current chip-multiprocessor (CMP) architectures is influenced by numerous factors including number of cores/threads in the implementation, core frequency application characteristics (e.g., arithmetic intensity), data locality, and cache topology. Because CMP architectures share resources among processing cores, the placement of threads or *thread affinity* can significantly impact the execution of a multi-threaded workload. On one hand if two threads contend for a particular shared resource (e.g., two floating point (FP) intensive threads running on the same hyper-threaded core) it can lead to performance degradation and increase power consumption. On the other hand, a favorable utilization of a shared resource (e.g., shared cache through inter-thread data locality) can result in power-performance benefits. Consider the results of running a parallel workload on an Intel Quad-core system as presented in Figure 1. The workload consists of four parallel applications, executed with five different affinity configurations. The numbers reported are normalized with respect to the default OS-enforced affinity. Significant variations in power-performance are observed for the different configurations. Although the best choice for power and performance coincides with configuration *aff1*, the choices diverge for subsequent points. This makes it imperative that the scheduler considers power-performance trade-offs when making thread placement decisions.

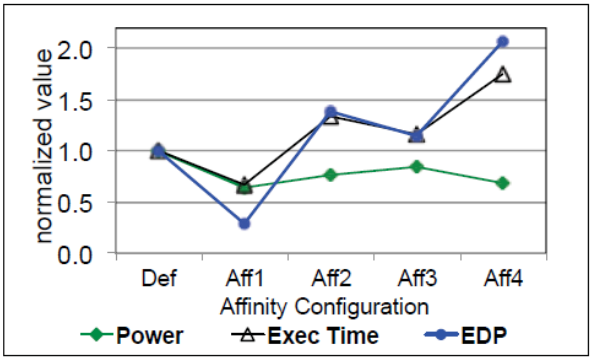


Figure 1. Impact of thread affinity on power, execution time and EDP

Although thread affinity is an important factor in power-performance-based optimizations, developing a scheduling heuristic that works well for different workloads is a challenging task. The difficulty arises from three main sources:

1) *Characterization of resource usage:* The demand and utilization of resources varies across workloads and across programs in a given workload. An intelligent scheduler needs a

 ISBN: 978-1-61208-432-9

mechanism that allows monitoring resource usage across the system and characterizing the utilization as either favorable or harmful. Characterization can involve determining inter-thread and intra-thread data locality and arithmetic intensity. This proves particularly problematic for an OS-level scheduler as it needs to extract information without the advantage of source code analysis. Furthermore, the scheduler has to relate resource usage to overall power consumption, a task that is particularly difficult because of the lack of availability of dynamic power consumption data.

2) *Dynamic and fine-grain system monitoring:* The scheduler must collect dynamic measurements at a fine granularity; generally at an interval that coincides with an OS-enforced scheduling quanta (referred to as *slice*). Moreover, this task has to be performed in a non-intrusive manner in order to reduce the impact on the execution of the workload.

3) *Inferring patterns in execution:* In many situations, a small change in the system does not necessitate a change in the placement policy. For example, transient processes often occupy cores for short periods without a major impact on the overall execution time or energy efficiency of the workload. The scheduler needs to be aware of such artifacts and make placement decisions only when a broad pattern change has been detected. For enforcing such a policy there is a need for dynamic feedback as well as for a method of maintaining historical data from which, execution patterns can be inferred.

In this work, we describe an affinity-driven meta-scheduler that addresses each of the above issues. The scheduler operates in the user-space as a runtime system and works in concert with the operating system. Central to our approach is the systematic and extensive utilization of hardware Performance Monitoring Units (PMUs). Today's commodity processors feature a large collection of PMU counters and registers with advanced capabilities that can provide a wealth of information about system performance. Although the use of PMUs is commonplace in application performance tuning in the high performance (HPC) domain, their use in operating system tasks such as mapping and scheduling is very limited. We leverage these performance counters and construct a set of models that synthesizes PMU counters values to provide insight into performance and energy related issues arising from the execution of a multi-threaded workload. Specifically, we use our PMU-based synthesized metrics to detect performance bottlenecks and power anomalies caused by contention of a shared resource or the under-utilization of a private (exclusive to a core) resource or computational units. We include, in the system, a mechanism to probe the PMU counters, derive the synthesized metrics at fixed intervals, and maintain historical data. The collected data is used as feedback for a novel greedy heuristic-based scheduling algorithm that makes dynamic affinity decisions to improve energy-efficiency and performance. Additionally, the framework facilitates the generation of training data and the integration of machine learning algorithms in scheduling decisions.

This paper makes the following contributions:

- It provides a low-overhead dynamic mechanism for detecting resource contention, sharing, and under-utilization on current multicore architectures.
- It provides new insight about power consumption and utilization of resources and demonstrates how these relationships can be exploited using a novel affinity-based power-aware scheduling heuristic.
- It presents the implementation of a portable meta-scheduler for Linux whose installation requires no modification to kernel code and no instrumentation of application binaries.

The remainder of this paper is organized as follows: Section II discusses related power-aware thread scheduling work. Section III provides an overview of the meta-scheduler framework. Section IV presents the synthesized metrics for resource characterization. Section V presents experimental results and Section VI includes conclusions and proposals for future research.

II. Related Work

Much of the work in scheduling for power has focused on developing runtime strategies that aim to find an optimal schedule for a single parallel application [1][5][8][11]. General strategies for power-aware scheduling are less common [4][7].

Bautista *et al.* present a power-aware scheduler that aims to minimize power consumption while respecting task deadlines in real-time applications [2]. Wierman *et al.* provide theoretical bounds on dynamic voltage and frequency (DVFS) based scheduling techniques [10]. They show that in terms of performance and power, a static DVFS scheduling strategy works as well as a dynamic strategy. However, a dynamic strategy can yield benefits when the objective is to improve system reliability. Kashif *et al.* propose a Priority-based Multi-level Feedback Queue Scheduler (PMLFQS) for mobile devices. PMLFQS is a work-conserving algorithm that uses different central processing unit (CPU) speeds to minimize the overall energy consumed by the CPU for each task [6].

Zong *et al.* have proposed two scheduling algorithms for scheduling parallel applications on large clusters. Their framework utilizes a precedence-constrained task graph of the application to be scheduled and emits a schedule that is predicted to be most energy-efficient [11]. Teodorescu et al. present a power management algorithm that takes into account variations in voltage and frequency among cores and attempts to improve performance within a given power envelope [9].

Merkel et al. develop heuristics that schedule threads according to a resource sharing utility. They combine these algorithms with DVFS techniques and evaluate their strategy on a workload with homogeneous sharing patterns. They demonstrate significant reduction in the Energy Delay Product (EDP) [7]. Boyd-Wickizer et al. propose a technique that operates at the level of objects and migrate threads from core-to-core depending on the data structures they access [4]. Bringing threads closer to the data reduces memory latency [4].

 ISBN: 978-1-61208-432-9

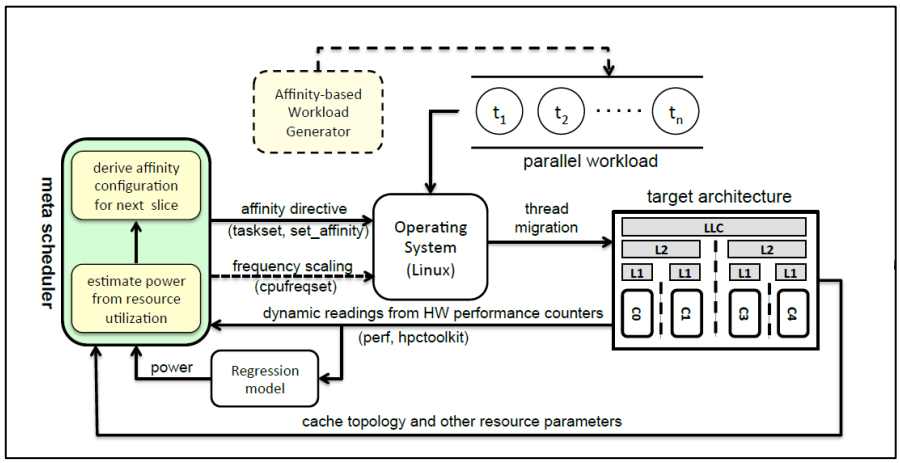


Figure 2. Meta-Scheduler Framework Overview

III. The META-SCHEDULER FRAMEWORK

Figure 2. Provides an overview of the framework. The central component is the meta-scheduler, which executes as a run time system. We have developed an API for communication between the meta-scheduler, the operating system, and the underlying hardware. The API allows the scheduler to probe and measure PMU counters, perform thread migration, and set core frequency. Special attention is given to ensure that each of these tasks are accomplished with low overhead. Next, we briefly discuss the implementation of each of these interfaces.

Hardware performance counters can provide detailed system information during workload execution. Software for probing this hardware has matured and there are several related tools such as *Intel Vtune*, *PAPI*, *HPCToolkit*, and *Oprofile*. Most of these tools, however, are primarily designed for single application tuning and those that do system-level profiling (e.g., *Oprofile*) carry significant overhead. Because our scheduler has to run at a low overhead we opted to implement our own interface that directly reads the PMU register values from device files.

The Linux *taskset* utility is used to set or change the affinity of a particular thread. The utility allows specifying the subset of cores that an application can be run on. It enforces a hard affinity on thread execution, which means that the OS always honors the affinity set by *taskset*.

It is difficult to determine dynamically in an inexpensive way if there is performance change in power demands. A run-time system that has access to PMU counters can monitor certain *trigger effects* that provide intuition into performance and power issues of running programs. This insight when modeled into a scheduling algorithm can yield significant gains.

In this section, we provide descriptions of three such resource-utilization metrics that are employed in our system. In the discussion that follows, we use the hardware counter names from the Intel Core micro-architecture. Similar counters are available on the AMD Barcelona and the IBM Power 7.

A. Core Utilization

Utilization of cores by different threads in a parallel system, is a key indicator of performance. Generally, a system running with a balanced load, where all cores are performing a similar amount of work, yields maximum concurrency and consequently leads to better performance. The utilization of cores has special significance when considering power consumption. With core gating it may be beneficial to consolidate the load into a subset of the cores while power gating the remaining cores. Regardless of the end objective, it is important to consider the workload balance while making thread mapping and scheduling decisions. These decisions, however, require accurate and dynamic measurements of core utilization at every time slice.

Current methods of measuring CPU utilization (e.g., Linux *top* utility) are not suitable for multicore architectures with multi-level caches and non-uniform memory accesses. In our framework, we use a set of counters to accurately estimate the amount of work done by each core. At each time slice we inspect the counters that provide the number of cycles a core is busy and the elapsed time (i.e., slice length). The number of elapsed cycles is a function of the clock frequency and elapsed time. However, because the core frequency can be modified during a given slice (e.g., via Intel's *Turboboost*), we use the *cpugovernor* to determine the current operating frequency of each core. The obtained frequency is used to determine the total number of elapsed cycles and the ratio of the busy cycles to elapsed cycles provides the core utilization.

B. Cache Behavior

The quality of cache utilization depends on intra and inter-core data locality. Yet, it is difficult to determine program locality without *a-priori* information. Nevertheless, by the examination of a set of PMU counters it is possible to determine favorable and non-favorable cache behavior.

 ISBN: 978-1-61208-432-9

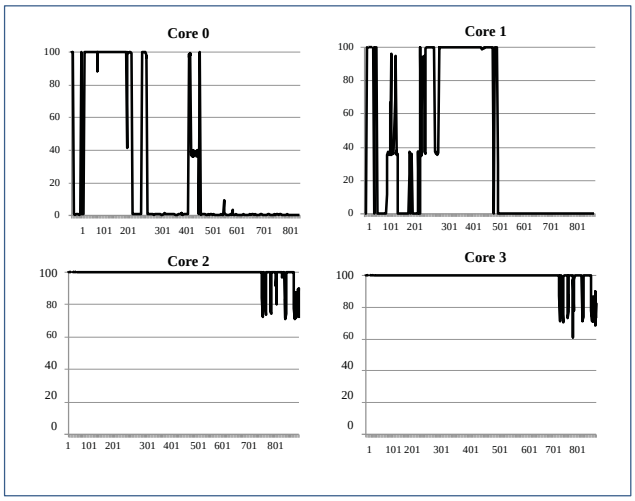


Figure 3. Per-core utilization for four-program workload broken down by core

If two threads share a cache and they have no shared data locality, then cache utilization is generally determined by their respective *working-sets* i.e., the amount of data accessed by each thread repeatedly. If the working set of the two programs exceeds the capacity of the shared cache then threads incur numerous misses in short succession. Applications do not necessarily access working sets during the entire execution. Therefore, the condition that needs to be checked is if both threads hit their respective working sets during the same time interval and exceed the capacity of the shared cache. This can be achieved by tracking per-core cache miss rates for the shared caches. There is contention in a shared cache if the average miss rate for shared cache in the last *k* intervals is significantly greater than the average miss rate of the same cache for the previous *j* intervals. A significant increase is determined by using a tolerance value as a tunable parameter. Since any applications can have multiple working sets that correspond to different caches, it is important to inspect contention at multiple levels.

Inter-core and inter-thread locality can have an impact on performance. If two threads have shared access to data and they are mapped to a set of cores that share cache then both execution time and power consumption benefits due to reduced cache misses. On the other hand, if two threads have no locality and they compete for cache space, then increase in cache misses reduces performance. Our system utilizes a set of performance counters to determine both favorable and unfavorable sharing of cache.

C. Computation Unit Utilization

System units can be shared at the hardware and thread level. For example, with hyper-threading two software threads running on the same core can share the FP unit. In this case it would be prudent to place threads that are FP-intensive onto different cores. Our system accounts for only one such resource, namely the on-chip FP units. However, as the system-on-chip (combined *GPU-CPU*) architectures become more prevalent, tracking utilization of other shared computational units will become more important.

V. EVALUATION

In this section, we present experimental results that illustrates the significance of resource characterization metrics in the energy-efficient execution of parallel workloads. We then show their effectiveness in making thread migration decisions with a greedy algorithm.

A. Experimental Setup

1) Platforms: the platform used is an Intel quad-core system that contains two Core 2 Duo processors sharing an L2 cache. Each core has a private L1 cache. The system runs Linux kernel 3.0. Under our adaptive thread migration policy the *cpugovernor* is set to custom. This eliminates interfere with the heuristics. When using the Linux strategy the *cpugoverner* is set to default.

2) Benchmarks: We evaluate our strategy on a variety of workloads generated from the *PARSEC* benchmark suite [3]. The suite includes a collection of multi-threaded programs with varying demands for system resources and contains data-, task-, and pipelined parallel applications. Each workload is formed from a subset of the *PARSEC* applications.

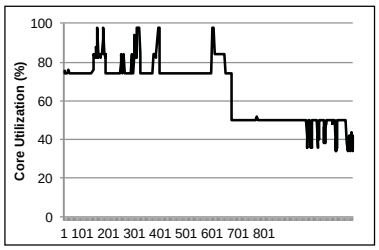
B. Load Balancing

First, we examine the way that workload characteristics and their affinity configurations impact the core utilization metric and the overall load balance of the system. Figure 3 shows the average core utilization of individual cores for *wkld1*, consisting of *canneal*, *streamcluster*, *blackscholes*, and *freqmine* [3]. The default migration policy of Linux is used in executing the workload. Significant variations in utilization of each core are observed throughout the execution of *wkld1*. In particular, cores 0 and 1 exhibit poor utilization during the first 400 time slices, while remaining under-utilized towards the end of the execution. Figures 4(a) and 4(b) show the average core utilization and the load balance of the system during four segments of execution. The system is close to a balanced state for only a small fraction of the time. More significant, however, is the fact that the average core utilization metric provides a clear indication concerning the balance of the system. The low utilization during times slices 300-400 would be a trigger for a smart scheduler and adjust the affinity to achieve better balance. Figures 4(c) and 4(d) present average core utilization and load balance information for a second workload (*wkld2*) that consists of *fluidanimate*, *canneal*, *streamcluster*, and *dedup* [3]. Interestingly, although *wkld2* contains two of the same applications as *wkld1*, we observe significant differences in the average core utilization. Even for this workload the average core utilization is a good indicator for the system load balance.

C. Cache Behavior

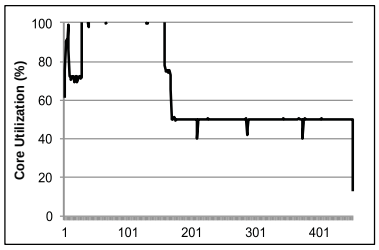
Figures 5 and 6 show last level cache (LLC) misses for *wkld3*, which consists of *raytrace*, *swaptions*, *streamcluster*, and *dedup* [3]. We observe that for both affinity configurations there is considerable fluctuations in the cache miss rates. These fluctuations, however, do not follow the same pattern for the two different configurations.

 ISBN: 978-1-61208-432-9

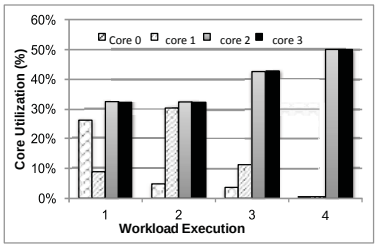


(a) Average core utilization *wkld1*

(b) System load balance for *wkld1*



(c) Average core utilization *wkld2*



(d) System load balance for *wkld2*

Figure 4. Variations in average core utilization and load-balance

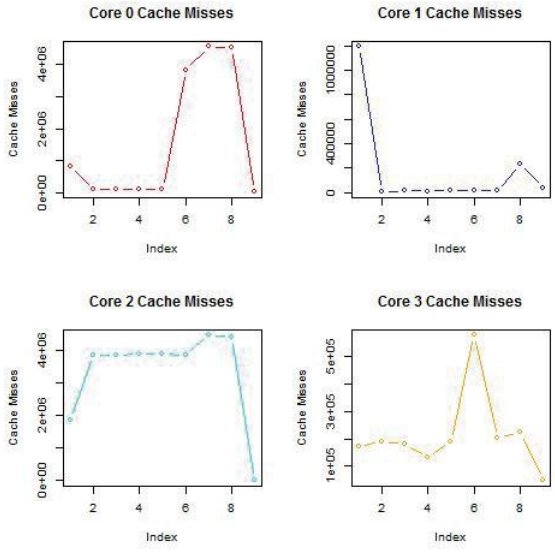


Figure 5. Core-level breakdown of LLC misses for *wkld2* with default affinity

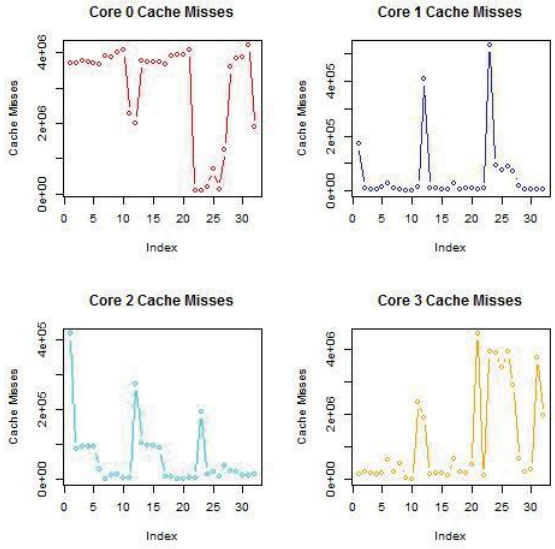


Figure 6. Core-level breakdown of LLC misses for *wkld2* with affinity configuration, aff2

This demonstrates the way that the cache miss rate can be impacted by the choice of affinity. The most interesting aspect of these results are the spikes in cache miss rates observed at various intervals (e.g., core 3 for *wkld2* at interval six). These sudden spikes can have a negative impact on performance and power consumption. To ameliorate the ill-effects of these spikes, the scheduler must have information at time slices boundaries, as provided by our framework.

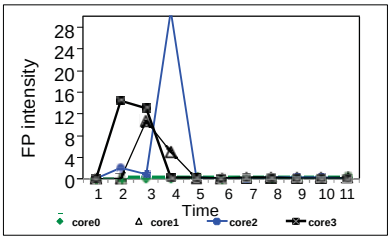
D. Computational Units

Figure 7 shows variations in arithmetic intensity for different affinity configurations for *wkld4*, which consists of *canneal*, *streamcluster*, *facesim* and *x264* [3]. Considerable variation in arithmetic intensity exists when different affinity policies are used. The *aff3* strategy shows the most balance in distribution of FP operations across the cores. Both *aff1* and *aff2* produce several spikes in FP-activity on core 0. For the default affinity, most FP operations are packed towards the beginning of the workload execution; but they are under-utilized later on.

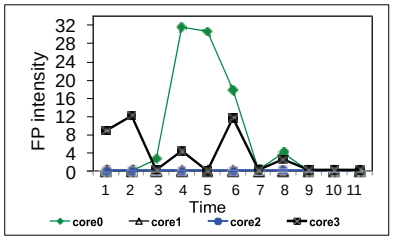
E. Evaluation with a Greedy Algorithm

We have implemented an adaptive thread migration algorithm that exploits resource characterization metrics and makes decisions based on a greedy heuristic. At each time slice the algorithm inspects the system, collects measurements, and tracks all currently running processes. A history table is used to store the resource usage data from the last *k* intervals. At each time slice the algorithm makes a decision on whether to change the current affinity in order to improve the load balance, cache sharing, and FP unit utilization. All decisions are weighed against the predicted power consumption for the next time slice.

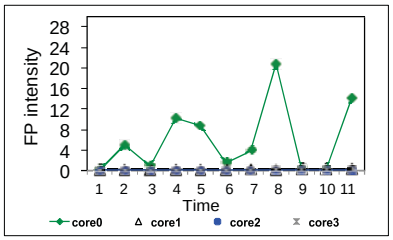
 ISBN: 978-1-61208-432-9



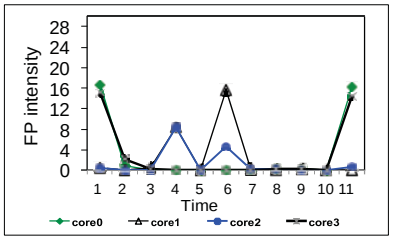
(a) Default Affinity



(b) Affinity 1



(c) Affinity 2



(d) Affinity 3

Figure 7. Variations in arithmetic intensity for different affinity configurations *wkld4*

TABLE I. ENERGY EFFCIENCY WITH GREEDY HEURISTICS

Workload	Power (W)		Exec. Time (s)		Energy (K Joules)	
	Linux	Greedy	Linux	Greedy	Linux	Greedy
wkld1	38.37	27.98	355	797	13.62	22.30
wkld2	38.25	32.93	306	277	11.71	9.12
wkld3	39.34	25.26	307	204	12.07	5.15
wkld4	28.16	26.59	10	13	0.28	0.34

Table I presents results of applying our algorithm on the four different workloads discussed earlier. We observe that in almost all the cases, the greedy algorithm outperforms the Linux scheduler in terms of energy dissipation and execution time. Particularly compelling is the situation with *wkld3*, where our greedy heuristic results in a 35% reduction in power consumption. Overall, on average, the greedy heuristic yields a 2% reduction in energy, 22% reduction in power consumption, and 32% increase in execution time.

VI. CONCLUSIONS

This work presents an energy-efficient thread migration strategy that is based on characterization of resource usage. We have identified a set of synthesized metrics that provide key insight into the execution behavior of parallel workloads running on contemporary multicore architectures. The experimental results show that core utilization, cache contention, and use of FP units can impact the execution and power consumption in intricate ways. We develop a greedy algorithm that exploits these synthesized metrics to significantly outperform the Linux scheduler both in terms of performance and energy efficiency.

In the future we plan to further the research and evaluate several rescheduling mechanisms. Additional avenue is to include additional shared resources.

VII. ACKNOWLEDGMENTS

Financial support for this work was provided by the Semiconductor Research Consortium (SRC) under contract no. 2011-HJ-2156 and the National Science Foundation through awards nos. CNS-1305302 and CNS-1253292.

REFERENCES

[1] I. Ahmad, R. Arora, D. White, V. Metsis, and R. Ingram, "Energy-constrained scheduling of dags on multi-core processors," in S. Ranka, S. Aluru, R. Buyya, Y. C. Chung, S. Dua, A. Grama, S. K. S. Gupta, R. Kumar, and V. V. Phoha, editors, Contemporary Computing, volume 40 of Communications in Computer and Information Science, pp. 592–603. Springer Berlin Heidelberg, 2009.

[2] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato "A simple power-aware scheduling for multicore systems when running real-time applications," in the IEEE International Symposium on Parallel and Distributed Processing, pp. 1-7 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in the Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72-81, 2008.

[4] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in the Proceedings of the 12th conference on Hot topics in operating systems, pp. 21-22 2009.

[5] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A scalable locality-aware adaptive work-stealing scheduler," in the IEEE International Symposium on Parallel Distributed Processing, pp. 341-342 2010.

[6] M. Kashif, T. Helmy, and E. El-Sebakhy. "A priority-based mlfq scheduler for CPU power saving," in the Proceedings of the IEEE International Conference on Computer Systems, pp. 130-134, 2006.

[7] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in the Proceedings of the 5th European conference on Computer systems, pp. 153-166, 2010.

[8] Q. Tang, S. K. S. Gupta, and G. Varsamopoulos, "Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach," IEEE Transactions on Parallel and distributed systems, vol. 19, pp. 1458–1472, 2008.

[9] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in the Proceedings of the 35th Symposium on Computer Architecture, pp. 353-374, 2008.

[10] A. Wierman, L. Andrew, and A. Tang. Stochastic analysis of power-aware scheduling," in the proceedings of the 46th Conference on Communication, Control, and Computing, 2pp. 23-26, 2008.

[11] Z. Zong, A. Manzanares, X. Ruan, and X. Qin, "Two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters," IEEE Transactions on Computers, pp. 360-374, 2009.

 ISBN: 978-1-61208-432-9