Dynamic Workload Characterization for Power Efficient Scheduling on CMP Systems[*]

Gaurav Dhiman[§]
gdhiman@cs.ucsd.edu

Vasileios Kontorinis[§]
vkontori@cs.ucsd.edu

Dean Tullsen[§]
tullsen@cs.ucsd.edu

Tajana Rosing[§]
tajana@ucsd.edu

Eric Saxe[†]
eric.saxe@oracle.com

Jonathan Chew[†]
jonathan.chew@oracle.com

[§]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

[†]Oracle Corporation
17 Network Circle
Menlo Park, CA 94025

ABSTRACT

Runtime characteristics of individual threads (such as IPC, cache usage, etc.) are a critical factor in making efficient scheduling decisions in modern chip-multiprocessor systems. They provide key insights into how threads interact when they share processor resources, and affect the overall system power and performance efficiency. In this paper, we propose and implement mechanisms and policies for a commercial OS scheduler and load balancer which incorporates thread characteristics, and show that it results in improvements of up to 30% in performance per watt.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Scheduling

General Terms

Design, Experimentation, Performance

Keywords

Power, Multi-cores, Workload Characterization

1. INTRODUCTION

With recent advances in processor architecture and the advent of chip multiprocessors (CMPs), also called multicore architectures, parallelism has become pervasive in modern systems. CMPs bring a hierarchy of levels of asymmetric resource sharing, resulting in a system where efficient scheduling of threads becomes non-trivial. Two threads scheduled on the same multithreaded core share pipelines, L1 caches etc. Two threads on the same multicore processor may share lower-level caches and various levels and partitions of the interconnect. Threads scheduled across multiple multicores still share off-chip buses, caches, and main memory. Thus, the ways in which these threads interact and impact each others' performance and overall power efficiency is critically affected by their *relative* placement on the shared execution resources.

This paper examines a modern Operating System (OS) running on a modern multicore architecture which exhibits such resource sharing asymmetries. It shows that the OS schedulers struggle to properly handle them, despite being specifically structured to do so. They fail to extract the full efficiency available from these parallel architectures, hence delivering poor performance per watt (Perf/Watt). We show that the primary reason for this is the lack of knowledge available to the scheduler and load balancer regarding the characteristics of the threads sharing these different resources. To solve this problem, we first identify and understand the correlation between the characteristics of the threads and the way these affect overall performance and power efficiency when they share resources. Based on this, we propose and implement mechanisms and metrics in a real OS for runtime workload characterization of threads. We then extend the scheduler and load balancer with policies to exploit this information to extract higher power and performance efficiency gains from the system. We refer to the new scheduler as the "Workload Characteristics Aware" (WCA) scheduler.

Furthermore, this paper also uncovers and provides a solution to a significant impediment to predictable scheduling – *transient threads*. These are threads that come onto the system for far too short a period (eg. kernel daemons) to have a direct impact on performance; yet mislead the scheduler about the load on the system, resulting in unnecessary load balancer induced thread migration decisions. These decisions can be pathologically bad, and result in non-deterministic run-times of threads and poor system level Perf/Watt. We show that transient thread characterization makes the WCA scheduler more efficient, and results in more predictable performance and overall power efficiency.

Based on this discussion, this paper makes the following *primary contributions*: (1) It presents the first implementation, together with power and performance results, for a commercial OS scheduler and load balancer with runtime workload characterization and feedback. Our results across 30 workloads indicate that it can improve the overall average Perf/Watt of the system (at negligible overhead) by 15% (maximum gain of 30%). (2) It identifies *transient threads*, initiated by modern OSs, as a critical impediment to properly characterizing threads and implementing predictable scheduling, and provides a solution to the problem.

The rest of the paper is organized as follows. Section 2 discusses prior related work. We then present the current state of the art in

[*]This work has been funded in part by Sun Microsystems, UC MICRO, Center for Networked Systems (CNS) at UCSD, MARCO/DARPA Gigascale Systems Research Center, NSF grant CCF-0702349 and NSF Greenlight.

scheduling methodologies and motivate our work in 3. The implementation details of our design are discussed in section 4 followed by details on our evaluation, methodology and results in section 5. We conclude in section 6.

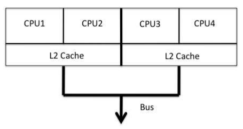
2. RELATED WORK

Significant prior research has sought to minimize conflicts and capacity problems on shared resources in a processor or multiprocessor (resources like last level cache, pipeline, etc.). Snavely, et al. [10] propose algorithms for efficient thread co-scheduling in multi-threaded processors, and demonstrate the benefits of symbiotic thread co-scheduling – this is a sampling-based solution, which can thus only reason about schedules that have actually been run. McGregor, et al. [6] show that on multiprocessors consisting of multiple simultaneous multithreading (SMT) cores, memory bandwidth utilization, bus transaction rate, and processor stall cycles of threads are as important as cache interference for determining the best co-schedules. Similarly, Bulpin, et al. [1] use hardware performance counters to derive a model for symbiotic co-scheduling on SMT processors.

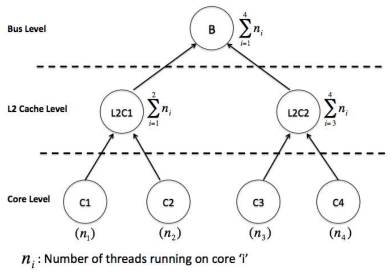
However, on a real system with dynamic workloads, run-time workload characterization is required to find schedules that minimize resource contention. Towards this end, Fedorova, et al. [4] propose a new design for an OS scheduler to reduce cache conflicts and capacity misses based on a model of cache miss ratios using CPU performance counters. However, their prototype is based on a user-level scheduler. Similarly, Zhang, et al. [11] propose the use of CPU performance counter as a first class system resource to dynamically detect and resolve resource contention. Knauerhase, et al. [5] and Merkel, et al. [8] propose schemes for minimizing resource contention at the last level cache (LLC) in a CMP system. They leverage the CPU performance counters to dynamically characterize the threads in terms of their LLC intensity, and modify the OS scheduler runqueue management mechanism to make sure that threads with heterogeneous cache usage characteristics are co-scheduled across the different cores at any given point in time. This provides speedup in multi-program workloads with heterogeneous characteristics. Merkel, et al. [8] show that this is beneficial from an energy efficiency perspective as well, since it results in a reduction in the energy required to execute a workload.

However, these approaches work only if there are enough threads in the system to fill up the runqueues of all the cores in the system, since otherwise the system is considered to be balanced. They also require threads with heterogeneous characteristics to be present across all the runqueues, since they sort each runqueue separately based on the cache intensiveness of the threads. With increasing core density in CMP processors, we are more likely to see scenarios where the total number of active threads in the system is less than or equal to the number of contexts in the system. State of the art load balancing algorithms in modern OSs like Linux and Solaris just balance the number of threads across the core resources, and are agnostic to their characteristics. Our work shows that this is largely insufficient and can result in thread schedules that cause severe resource contention and degrade overall system power and performance efficiency. We further highlight the instability introduced into the system in such scenarios due to short-running threads, which we refer to as *transient threads*.

In terms of active power management on modern CMP based systems, recent work [7, 8, 3] shows that the effectiveness of techniques like dynamic voltage scaling has largely diminished due to a combination of factors like increasing leakage, faster memory speeds etc. Instead, it is more energy efficient to run the system faster within a given set of workloads and then utilize low power

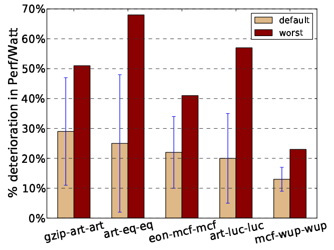


(a) Topology of an Intel Quad Core Xeon E5430 Processor

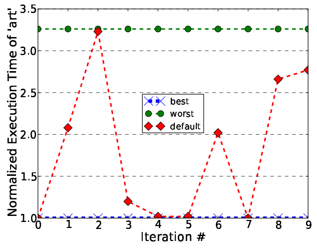


(b) Processor group structure

Figure 1: Example of a processor and its corresponding processor group constructed by OpenSolaris



(a)



(b)

Figure 2: (a) % deterioration in Perf/Watt normalized against the best case; (b) Normalized run-times for "art" across 10 iterations in art-equake-equake combination

modes available widely on modern systems [7, 3, 8]. This observation motivates our approach towards power efficiency. We exploit the workload characteristics to reduce resource contention, and achieve higher performance within roughly the same power budget, thereby significantly improving the overall Perf/Watt of the system.

3. BACKGROUND AND MOTIVATION

Modern processors are comprised of multiple computing cores that share different resources (like pipelines, caches etc.) across the system. The degree of sharing is often asymmetrical based on the core location. For instance, Figure 1a illustrates the layout of a state-of-the-art Intel Xeon quad core E5430 processor. Cores 1 and 2 share an L2 cache, as do cores 3 and 4, while all cores share the same bus going to the memory. This means that in terms of sharing the bus, the relationship between any two cores in this processor is symmetrical, while in terms of sharing L2 cache they are not.

It is imperative that the OS scheduler capture this sharing relationship among the different computing cores, and ensure that the resource conflicts are minimized. Most modern OSs do this by building abstractions dynamically to capture the hardware topology of the system, and then load balancing the threads across the topology to minimize resource contention. Examples of such abstractions include processor and logical groups in OpenSolaris, and scheduler domains in Linux. In this paper, we will focus on the former, though the basic underlying principles are similar for both.

Figure 1b gives an example of a processor group (PG) structure that OpenSolaris would build for the processor shown in Figure 1a. It is a hierarchical structure, where each level represents a shared resource. The computing cores form the leaves of the structure, and have their own pipelines and L1 caches (referred to as $c1$, $c2$ etc). The next level represents the L2 cache, and the cores sharing them are joined together into one node to represent the sharing (referred to as $L2C1$, $L2C2$). The next level represents the bus, which is shared by all the cores (referred to as B). The load balancing algorithm seeks to ensure that the number of running threads at any point in time are balanced across nodes at all levels. It does so by maintaining a count on the number of threads at each level of the hierarchy (referred to as n_i in Figure 1b), and balances it by thread migrations across different cores. This is our baseline/default scheduler.

To understand the performance of the default scheduler we did some offline experiments with benchmarks with different instruction and memory characteristics from the SPEC2000 benchmark suite. We created three-program workloads with two instances of one benchmark and one instance of the other (eg. gzip-art-art). Such workloads provide only two possible schedules in terms of thread assignment to cores (eg. either 'gzip' and 'art' share cache or 'art' and 'art' share it). For each run of the workload we measured the execution time of the threads as well as the whole system level power consumption under three different schedules. In the first two, we manually create the best and worst among the two possible schedules in terms of Perf/Watt by binding threads to cores. In the third, we let the default scheduler determine the schedule.

Figure 2a shows the results for the worst and the default schedules (relative to the best) averaged across 10 runs for 5 workloads. The large gap between the best and worst schedules highlights the importance of identifying good schedules, and the fact that the actual default scheduler falls roughly in the middle, with a high variance, demonstrates its inability to distinguish between good and bad schedules. For instance in the case of the art-equake-equake combination, the Perf/Watt degrades by as much as 70%. This happens because art is very sensitive to its L2 cache data, and when it shares it with equake (in the worst schedule), its execution time goes up significantly.

The results for the default scheduler come from ping-ponging between the best and worst schedule, both across different runs but also within a run. This is illustrated by the error bars in Perf/Watt in Figure 2a which indicate the variance in the measured run-times. This variance is further highlighted in Figure 2b, which shows 'art' performance for the various runs in the art-equake-equake workload. The runtime variance is explained by the presence of kernel high-priority threads that regularly preempt the long-running threads during the experiment. We refer to these short, high priority running threads as *transient threads*. Transient threads, as we show in the next section, result in unnecessary load balancer induced thread migrations, which result in the thread schedule of the workload oscillating between best and worst at a random frequency. Running these same experiments on a Linux system, we observed the same type of runtime variance across different runs, with Perf/Watt degradation as high as 70%. This issue is not specific to a particular OS, then, but represents a much more general problem – insufficient abstractions in modern OSs to account for the nature of the load in the system, beyond simple thread counts.

Thus, these experiments motivate both of the key findings of this paper: (1) We must provide the scheduler with more information to better balance resources (in these experiments, cache utilization is the dominant factor) and identify good schedules for extracting higher Perf/Watt from the system, and (2) We must account for the existence of short-running transient threads in the scheduler to avoid frequent disruption of good schedules.

4. DESIGN AND IMPLEMENTATION

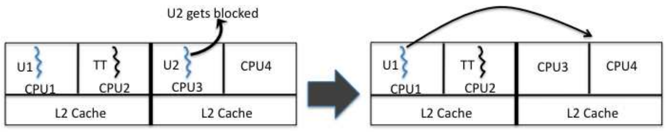
Based on the discussion in the prior sections, we need characterization of threads on two fronts: (1) **Transience** captures the amount of time a thread stays blocked relative to the time it spends executing. (2) **Cache sensitivity** captures the sensitivity of a thread's performance to a cache, and contention for that cache. This section details the design principles behind measuring and modeling these characteristics.

4.1 Thread Transience

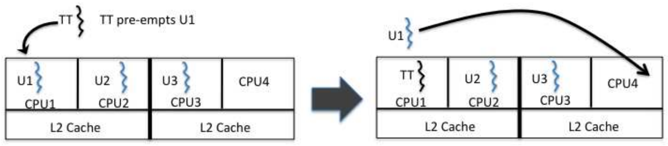
In a running system there are many kernel threads that are used to service various tasks in the system. These threads typically stay blocked the vast majority of the time and then run for a very short duration (on the order of microseconds). Examples of such threads that we observed in OpenSolaris included java, fsflush (daemon for flushing dirty file system pages), nscd (caching daemon), etc. Being kernel threads, they have high priority and get to run as soon as they become runnable, possibly pre-empting a user thread. Since they run for such short amounts of time, they have little direct impact on the overall execution time of the user threads they pre-empt in terms of either CPU time or cache footprint . However, the problem is that the OS load balancer is agnostic to the runtime characteristics of threads; as a result, it cannot distinguish transient threads. It strives to load balance for these threads, making decisions that may be highly detrimental to overall system efficiency.

Figures 3a and 3b show two scenarios where transient threads can result in unnecessary load-balancer-induced migrations. In scenario 1, thread U1 and U2 are user threads running on CPUs 1 and 3 when a transient thread TT becomes runnable. However, as soon as TT gets scheduled on CPU1, U2 gets blocked for disk I/O as shown in Figure 3a. At the same time, the OS load balancer is invoked and it detects an imbalance in the PG nodes $L2C1$ and $L2C2$ (refer to figure 1b), since there are two threads under one, and none under the other. Thus, it migrates U1 to either CPU3 or 4 to restore balance. However, this is an unnecessary migration, since TT may again become blocked before thread U1 even fully completes its migration. We refer to this scenario as the 'artificial load' case, since the load balancer is made to believe there is load in the system when there practically is none. In scenario 2, three user threads U1-3 are running. A transient thread TT becomes runnable and pre-empts U1, as shown in Figure 3b. This can happen due to the fact that TT last ran on CPU1, and hence according to the scheduler has cache affinity for it. Now the idling CPU4 will detect U1 in the CPU1 runqueue and will steal it for better balance and fairness. However, this is also an unnecessary migration. We refer to this scenario as migration from an 'almost idle' CPU – we are migrating load away from a CPU that is going to become idle almost immediately.

The solution to the transient thread problem is a simple two-part process. *First*, we identify transient threads. To identify transient threads, we make a simple observation: they spend the majority of their time blocked. For this purpose, we add a flag to indicate whether a thread is transient or not, and maintain a runtime history that captures the time it spends on and off the CPU in the OpenSolaris thread data structure. Based on the ratio of "on time" to the total time, we mark the thread as transient if the ratio falls below a threshold $T_{On_{tr}}$ and vice versa. In our experiments, the value of $T_{On_{tr}}$=1% served us well in cleanly separating the transient threads from the non-transient ones. *Second*, we modify the load balancing algorithm of OpenSolaris to balance only the non-transient threads across the system. For the 'artificial load' problem, Figure 3a, this means that the load balancer would no longer consider the system to be imbalanced when U2 gets blocked, since only one non-transient thread (U1) is there. In the 'almost idle' case, Figure 3b, CPU4 would not steal U1, since there is only 1 non-transient thread on CPU1.



(a) Scenario-1 (Artificial Load)



(b) Scenario-2 (Migration from 'almost idle' CPU)

Figure 3: Examples of avoidable migrations due to transient threads

Thus, a transient thread characterization aware scheduler has a much more accurate view of the real load on the system, and as we show in Section 5, makes the system more efficient and deterministic.

4.2 Cache Sensitivity

Based on the observations in section 3 , we know that sub-optimal cache sharing can significantly degrade the performance of cache sensitive threads. To alleviate this problem, we need the OS load balancer to know the cache sensitivity of threads it is trying to schedule. This requires two steps: (1) Identify characteristics/metrics that separate a cache sensitive thread from a non-sensitive one, (2) Reconstruct the load balancer to use this information to more effectively balance the demand on shared resources. In this section we discuss the metrics we employ and how we extend the OpenSolaris load balancer to account for shared caches.

Metrics for cache sensitivity While prior work [10, 6, 11] has shown that a sophisticated model of the expected interactions of two threads that share a cache can create better schedules, a real OS needs a scheduler that is distributed, fast, and runs in constant time. Hence, we adopt a simple and more coarse-grained approach, which is both constant time and utilizes commonly available CPU performance counters for performing binary classification to distinguish among the cache-sensitive and cache-insensitive threads. Good candidates for a binary classification heuristic should exhibit low variance during the execution of a program and low sensitivity to the current schedule. Based on these criteria, we identify the following two as complementary and effective indicators of cache sensitivity of a thread: (1) *Last level cache requests per instruction* (LLCRPI): The threads with high LLCRPI are highly likely to have a big chunk of their working set in the LLC (last level cache). Thus, sharing it with another thread increases the probability of its useful data (and the other thread's) getting evicted. (2) *Instructions committed per cycle* (IPC): Threads with high IPC typically see a larger degradation when their working set is suddenly displaced by another thread. Although both of these factors affect co-scheduled performance, the LLCRPI (cache working set) is the dominant effect and is given priority over IPC. To account for this phenomenon we assign different weights of cache sensitivity to these two different categories of cache sensitive threads, with a higher weight signifying higher sensitivity and priority. Specifically, high LLCRPI threads (LLCRPI > 0.1) are attributed a cache weight (CW) of 2 while the high IPC (IPC > 1) threads are attributed a CW of 1. This ensures that we can further refine our scheduling decisions when all applications are cache-sensitive, so that high LLCRPI benchmarks are preferred to be scheduled alone over the high IPC ones. All the other threads in the system are identified as cache insensitive (CW=0).

WCA scheduler Implementation: With the metrics for cache sensitivity defined, we need to dynamically capture the required data, and use the classification for load balancing. We make use of CPU performance counters provided by the Intel Xeon performance monitoring unit. The counters that we use are: Instructions retired (INST), clock cycles (CLK) and LLC requests (LLCR). When-

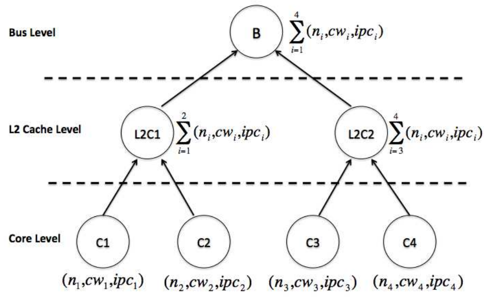


Figure 4: Modified PG structure

ever a thread begins to run on the CPU, we turn the performance counters on to monitor these events. When the thread leaves the CPU or completes its time slice, we calculate IPC_{cur} and $LLCRPI_{cur}$ for the latest run. However, these numbers represent the metrics only for the current run of the thread on the CPU. To account for longer-term effects, we maintain them as exponential averages as shown below:

$$\begin{aligned} IPC_{new} &= \alpha \cdot IPC_{cur} + (1-\alpha) \cdot IPC_{prev} \\ LLCRPI_{new} &= \alpha \cdot LLCRPI_{cur} + (1-\alpha) \cdot LLCRPI_{prev} \end{aligned} \quad (1)$$

where the updated average (eg. IPC_{new}) is the exponential average of the previous average (eg. IPC_{prev}) and the current value (eg. IPC_{cur}). The factor α represents the relative weight of the previous and current value, which we set to 0.5 to give equal weight to both. Based on these average metric values we determine the cache weight (CW) of a thread as described in the discussion on metrics previously. These metric estimates and updates occur independently on each of the CPUs in the system in a distributed fashion.

To incorporate these characteristics into the load balancing fabric, we modify the PG structure to include these metrics. Figure 4 shows the modified PG for the Intel Xeon E5430 processor. A quick comparison to the original structure in Figure 1b indicates that now at each level of the hierarchy we maintain a 3-tuple of (number of threads, cache weight, IPC) instead of just the number of threads. Cache weight (CW) takes into account the cache sensitivity of threads based on their LLCRPI and IPC. For the case where the cache weight of all running threads are the same, we further store the absolute IPC as our tiebreaker – we then determine the best schedule by giving priority to the high IPC thread. This is based on the observation that a higher IPC thread is likely to undergo higher performance loss when sharing cache.

Table 1 shows our modified load balancing algorithm, which balances all these parameters across the PG hierarchy for minimizing resource contention. The algorithm first tries to balance the number of threads across nodes at different levels in Figure 4. For the Intel Xeon E5430 processor, this corresponds to balancing the number of threads under $L2C1$ and $L2C2$. The algorithm is invoked independently on each CPU, and runs in the context of the thread sched-

Table 1: WCA Scheduler Load Balance Algorithm

```
$T \leftarrow curthread$
$N_1 \leftarrow L2C1$
$N_2 \leftarrow L2C2$
/*Assume T is running on CPU under L2C1*/
 1: if $(N_1 - 1) > N_2$ then
 2:     /* Stage 1: Balance number of threads */
 3:     Migrate $T$ to an idle CPU in $N_2$
 4:     return
 5: end if
 6: /* The default load balancer ends here */
 7: if $(CW_{N_1} - CW_T) > CW_{N_2}$ then
 8:     /* Stage 2: Balance cache weight */
 9:     Migrate $T$ to an idle CPU in $N_2$
10:     return
11: end if
12: if $(IPC_{N_1} - IPC_T) > IPC_{N_2}$ then
13:     /* Stage 3: Balance IPC */
14:     Migrate $T$ to an idle CPU in $N_2$
15:     return
16: end if
```

uled on it. Lets assume that the algorithm is invoked for a thread T, which is running on CPU1 (i.e. under $L2C1$). Further, let N_1 and N_2 be the pointers to the L2 cache level PG structures $L2C1$ and $L2C2$ respectively. The algorithm first of all checks if there is an imbalance in terms of the number of threads running under the two PG siblings (N_1 and N_2). This could happen if there are two threads running under N_1, and none under N_2. If that is the case, then the thread T migrates itself to an idle CPU under N_2 (step 3 in Table 1). In the default load balancer, this is where the balancing finishes and the algorithm returns. However, this logic will fail to detect cache imbalances. For instance, consider a scenario where two cache sensitive threads are running under N_1, while one non cache sensitive thread is running under N_2. The load balancing algorithm in this case would consider the number of threads to be balanced, although severe cache contention results under N_1.

Step 7 and beyond in Table 1 represents the logic we add to the load balancing algorithm to exploit runtime workload characterization and address possible resource contention. In step 7, our algorithm checks for cache weight imbalance. If this is the case (like in the example above), thread T migrates itself to an idle CPU under N_2 as shown in step 9 in Table 1. The check in step 7 ($(CW_{N_1} - CW_T) > CW_{N_2}$) ensures that the migration should take place only if it makes the system more balanced and does not swap the imbalance between N_1 and N_2. Such a swap is undesirable since it can result in a ping-pong migration of the thread between N_1 and N_2. If this stage is also balanced, then it implies that both the number of threads and cache weight is balanced across the PG. At this point, the algorithm checks whether T is co-scheduled with a higher IPC thread compared to the one scheduled under N_2 (step 12 in Table 1). If that is the case, T migrates to an idle CPU under N_2. As discussed before, this ensures more fine grained balancing if the cache weight is already balanced across the PG.

5. EVALUATION

5.1 Methodology

We perform our experiments on a state of the art 45nm Intel quad core Xeon E5430 based server machine. The operating system is based on OpenSolaris build 98, which is modified to include our changes, outlined in Section 4. For our experiments we construct multi-program workloads using a selection of twelve SPEC2000 benchmarks. The benchmark subsetting is based on Tables 2 and 3 of [9], so that it covers all SPEC suite clusters according to their overall characteristics and data locality. The selected benchmarks were used to generate what we refer to as the 3-thread workloads. The 3-thread workloads are of the form of *bench1-bench2-bench2*, where $bench1$ and $bench2$ are different benchmarks. We choose 3-thread workloads, since it creates a difficult case for the scheduler from the perspective of both workload characterization as well as transient threads. The case of 1 and 2 thread workloads is uninteresting, since the long running threads will never share any resources (see Figure 1), while the 4-thread case does not expose the transient thread problem, since all the cores are occupied. In future many-core architectures many such combinations exposing both the issues would be possible. The choice of duplicated thread is intentional – it helps us understand the effect of *bench1* execution on *bench2* both with and without cache sharing. Across all the runs we record the execution times of the threads and measure the power of the whole system using a power analyzer at the granularity of 500ms.

To evaluate the effectiveness of our WCA scheduler against the default one for each workload combination, we make use of the average weighted Perf/Watt metric (based on a similar metric in [10]). The metric is defined as:

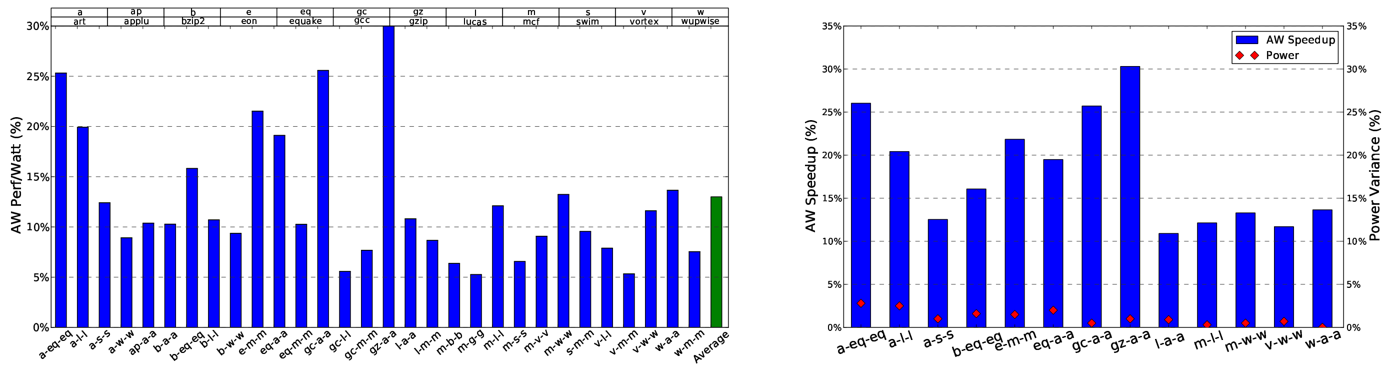
$$AWPerf/Watt = \left(\frac{\sum_{bench_i} \frac{T_{default_i}}{T_{alone_i}}}{\sum_{bench_i} \frac{T_{WCA_i}}{T_{alone_i}}}\right) / \left(\frac{P_{WCA}}{P_{default}}\right) - 1 \quad (2)$$

where $T_{default_i}$ and T_{WCA_i} are the execution times of a benchmark i in the workload combination with the default and our WCA scheduler respectively. These run-times are normalized against the execution time of the benchmark i when it is running alone in the system (without any contention). P_{WCA} and $P_{default}$ refer to the average system level power consumption for the benchmark combination run with the WCA and default scheduler respectively. A positive value of AW Perf/Watt indicates that WCA scheduler is more power efficient than the default scheduler (referred to as *gain*) and vice versa (referred to as *loss*).

5.2 Results

Average weighted Perf/Watt: This section demonstrates the gains in Perf/Watt achieved by our changes to the OpenSolaris scheduler. In the first set of experiments we ran workload combinations comprised of threads with heterogeneous characteristics (at least one cache sensitive and one not). Figure 5a illustrates the actual AW Perf/Watt values across 30 such workload combinations. We can observe that the WCA scheduler achieves significant gains, as much as 30% better than the default OpenSolaris scheduler, with an average close to 15%. The highest gain (30% in gzip-art-art) comes because the WCA scheduler is able to dynamically identify the high cache sensitivity of art and prevent the two instances from sharing a cache. The transient thread characterization makes the threads stick to the cores they have been scheduled on and thus preserve these gains. As a consequence, we observe that across all the workload combinations, the AW Perf/Watt results of the WCA scheduler are equal to the best possible (calculated offline by binding threads to cores in an optimal fashion).

Figure 5b illustrates the breakdown of the AW Perf/Watt results into AW Speedup and power variance ratio across 13 workloads with the highest gains. The strong gains in Perf/Watt are achieved with relatively unchanged power and significant throughput gains. This means we are able to process jobs faster with no corresponding rise in power – this results in significant energy savings per job, and accelerates the system to a condition where it can employ low-power states. Recent research suggest that this is a much more ef-



(a) (b)

Figure 5: (a) The AW Perf/Watt results; (b) Breakdown of AW Speedup and Power Variance ratio ($\frac{P_{WCA}}{P_{default}} - 1$) for workloads with highest AW Perf/Watt gains. The WCA scheduler is able to extract high speedups in roughly the same power budget.

Table 2: Rate of migrations and standard deviation of runtime normalized to best schedule over 10 different runs with default and the WCA scheduler

Workload	Default		WCA		Reduction(%)	
	mig/s	Std.Dev.(%)	mig/s	Std.Dev.(%)	mig/s	Std.Dev.(%)
art-equake-equake	1.01	86%	0.14	2%	87%	98%
mcf-applu-applu	1.46	21%	0.06	1%	96%	96%
bzip2-lucas-lucas	1.22	22%	0.04	1%	97%	96%
vortex-swim-swim	1.28	15%	0.10	2%	92%	87%
gcc-wupwise-wup.	1.67	12%	0.14	3%	91%	75%
art-mcf-mcf	1.42	38%	0.17	4%	88%	90%
mcf-bzip2-bzip2	1.42	4%	0.17	1%	88%	75%
vortex-equake-eq.	1.14	44%	0.05	3%	96%	93%
Average	1.33	30%	0.11	2%	91%	89%

fective way of performing energy efficient computation, compared to active power management techniques like DVFS [7, 3].

We next did experiments across workloads comprising of threads with homogeneous characteristics. We observed that for such combinations the overall performance of the WCA scheduler converged to that of the default scheduler, since the individual threads benefited little from characterization aware placement (the resource usage was homogeneous anyway). Thus, these set of experiments indicate that WCA scheduler is able to extract much higher efficiency out of the system compared to default scheduler, when threads with heterogeneous characteristics are present in the workload, and deliver equivalent performance, if that is not the case.

Workload Performance Predictability: Besides performance and power efficiency, there are two other qualities we would like from our scheduler – Performance predictability (or stability) and reduced migrations. The result of excessive migrations can be seen both in Perf/Watt stability (results shown here) and raw efficiency (shown in Figure 5a). Transient thread identification dramatically reduces the number of total thread migrations on the system, as shown in Table 2. To measure this, we use DTrace [2] to instrument the OS code responsible for migrations.

Overall, we observe from Table 2 over 90% average reduction in the total number of migrations with the WCA scheduler. The high migration rate in the default scheduler results in high variance in its execution time across successive runs, highlighted by an average standard deviation of 30% in Table 2. In contrast, that variance is dramatically reduced with the WCA scheduler (around 90%). A fine-grained analysis of the migrations with the WCA scheduler shows that the number of migrations that still happen in the system are all due to non-transient threads. Therefore, the transient thread characterization virtually eliminates avoidable migrations, and achieves predictable and power efficient run-times for the workloads across multiple runs by making sure the schedules determined by the WCA scheduler are adhered to.

6. CONCLUSIONS

This paper demonstrates modifications to an existing OS scheduler, already tuned to identify and balance across shared resources, that significantly improve power and performance efficiency. It does so by incorporating lightweight runtime workload characterization into the scheduler and extending the load balancer to exploit them to better balance demand for shared resources. It also shows that it is critical that we characterize transient threads to prevent them from causing the load balancer to overreact to short-lived imbalances for more efficient and stable schedules. The real life implementation of our framework achieves up to 30% improvement in performance per watt over the default OpenSolaris scheduler, and reduces thread migrations by close to 90%.

7. REFERENCES

[1] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proc. ATEC*, 2005.

[2] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. ATEC*, 2004.

[3] G. Dhiman, K. Pusukuri, and T. Rosing. Analysis of dynamic voltage scaling for system level energy management. In *USENIX HotPower*, 2008.

[4] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. ATEC*, 2005.

[5] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *Proc. IEEE Micro*, 28(3), 2008.

[6] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *proc. IPDPS*, 2005.

[7] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *Proc. ASPLOS*, 2009.

[8] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *USENIX HotPower*, 2008.

[9] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *Proc. ISPASS*, 2005.

[10] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Proc. ASPLOS*, 2000.

[11] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proc. USENIX HOTOS*, 2007.