Portable Performance on Asymmetric Multicore Processors

Ivan Jibaja[†*] Ting Cao[δψ] Stephen M. Blackburn[ψ] Kathryn S. McKinley[‡]

[†]The University of Texas at Austin, USA [*]Pure Storage, USA [δ]SKL Computer Architecture, ICT, CAS, China
[ψ]Australian National University, Australia [‡]Microsoft Research, USA

ivan@cs.utexas.edu, caoting@ict.ac.cn, steve.blackburn@anu.edu.au, mckinley@microsoft.com

Abstract

Static and dynamic power constraints are steering chip manufacturers to build single-ISA Asymmetric Multicore Processors (AMPs) with big and small cores. To deliver on their energy efficiency potential, schedulers must consider core sensitivity, load balance, and the critical path. Applying these criteria effectively is challenging especially for complex and non-scalable multithreaded applications. We demonstrate that runtimes for managed languages, which are now ubiquitous, provide a unique opportunity to abstract over AMP complexity and inform scheduling with rich semantics such as thread priorities, locks, and parallelism—information not directly available to the hardware, OS, or application.

We present the WASH AMP scheduler, which (1) automatically identifies and accelerates critical threads in concurrent, but non-scalable applications; (2) respects thread priorities; (3) considers core availability and thread sensitivity; and (4) *proportionally* schedules threads on big and small cores to optimize performance and energy. We introduce new dynamic analyses that identify critical threads and classify applications as sequential, scalable, or non-scalable. Compared to prior work, WASH improves performance by 20% and energy by 9% or more on frequency-scaled AMP hardware (not simulation). Performance advantages grow to 27% when asymmetry increases. Performance advantages are robust to a complex multithreaded adversary independently scheduled by the OS. WASH effectively identifies and optimizes a wider class of workloads than prior work.

Categories and Subject Descriptors D.3.4 [*Processors*]: Run-time environments

Keywords Scheduling, Asymmetric, Multicore, Heterogeneous, Managed Software, Performance, Energy

1. Introduction

For decades, faster processors delivered hardware performance improvements transparently to software, allowing software and hardware to innovate independently. To address technology constraints, vendors introduced multicore processors and recently single-ISA Asymmetric Multicore Processors (AMPs), which have the potential to significantly increase performance under static and dynamic power constraints [10, 13, 19]. The more transparent system designers make this complex evolving hardware to software developers, the better for software and hardware innovation.

Hardware trends AMPs combine *big* high-performance cores and *small* energy-efficient cores to improve performance and energy, while meeting power and area constraints [10, 13]. Big cores accelerate critical latency-sensitive threads, while numerous small cores deliver throughput for parallel workloads and energy efficiency for non-critical tasks. AMPs are already in mobile systems [19] and expected in desktops and servers. Because extracting performance from AMPs is hardware specific and further complicated by multiprogramming, application programmers cannot be required to manage this complexity nor to continuously adapt their software to specific and evolving hardware.

Challenges To achieve the performance and energy efficiency promises of AMP is challenging because the runtime must reason about *core sensitivity*, which thread's efficiency benefits most from which core; *priorities*, executing non-critical work on small cores and prioritizing the critical path to big cores; and *load balancing*, effectively utilizing available hardware resources. Prior work addresses some but not all of these challenges. For instance, prior work accelerates the critical path on big cores [8, 11, 12, 22], but needs programmer hints and new hardware. Other work manages core sensitivity and load balancing with proportional fair scheduling [6, 7, 20, 21] but is limited to scalable applications with equal numbers of threads and hardware contexts, an unrealistic assumption in a multiprogrammed context. Many real-world parallel applications also violate this assumption. For instance, the eclipse IDE manages logical asynchronous tasks with more threads than cores. Even if the application matches threads to cores, (1) some runtimes add compiler and garbage collection helper threads, and (2) many programs exhibit *messy* non-scalable parallelism. We experimentally show prior work performs inconsistently, especially on messy workloads.

A Managed Language AMP Runtime This paper targets managed applications. Managed languages are increasingly popular and thus important for mobile, desktops, and servers. We show managed runtimes offer a unique opportunity to transparently abstract over AMP hardware, because they already profile, optimize, and schedule applications.

Our AMP-aware runtime uses dynamic analysis to classify application behavior as single threaded, non-scalable multi-threaded, and scalable multi-threaded and then customizes its scheduling decisions accordingly. For instance, it proportionally fair schedules scalable applications and accelerates the critical path in non-scalable applications.

A key contribution of our work is a new dynamic analysis that automatically identifies *bottleneck threads* that hold contended locks and prioritizes them by the cumulative time other threads wait on them. None of the prior work automatically identifies and prioritizes threads. We show this analysis finds and accelerates the critical path, improving messy non-scalable workloads. For efficiency, we piggyback it on the VM's biased locking [1, 2]. Our VM profiling periodically monitors thread progress, thread core sensitivity, and communicates scheduling decisions to the OS with thread affinity settings.

Evaluation We implement our AMP runtime in a high performance Java VM for desktops and servers. Its mature compiler, runtime, and benchmarking ecosystems make it a better evaluation platform than immature mobile systems. We evaluate benchmarks from active open source projects on an AMD Phenom II x86 with core-independent frequency scaling (DVFS) configured as an AMP. Prior work establishes this methodology [5, 21], which dramatically increases experiments compared to simulation, which is slow and less accurate. On a range of AMP configurations, WASH improves energy and performance over prior work: 9% average energy and 20% performance, and up to 27% as hardware asymmetry increases on all workload types.

Figure 1 shows a sample result for messy non-scalable multi-threaded workloads on a three big / three small AMP configuration. It compares WASH to: (a) The default round-robin Linux scheduler [16, 18], (b) Proportional core-sensitive Fair Scheduling (PFS) [7, 21], the closest related work, and (c) bindVM [5], which simply binds VM helper threads to small cores and application threads to big cores. Lower is better. These results reveal that: (1) simple round robin scheduling is usually better than PFS because PFS does not identify the critical thread in messy workloads; (2) simply binding helper threads to small cores means critical application threads often get more resources, and thus bindVM often outperforms round robin; and (3) WASH identifies and prioritizes bottleneck threads that hold locks in messy non-scalable workloads with a small amount of overhead,

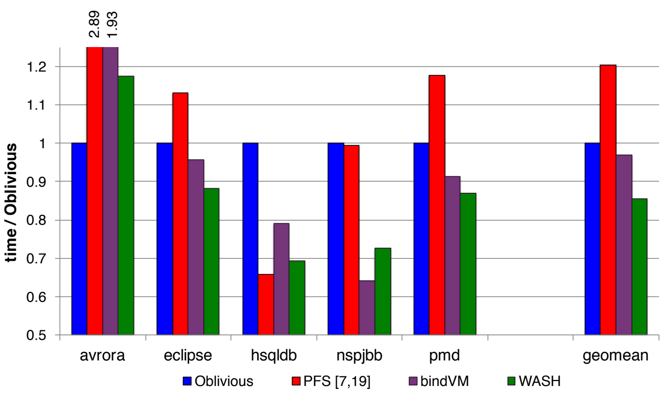


Figure 1: Messy non-scalable performance with WASH, Linux (Oblivious), PFS, and bindVM on three big / three small AMP. All prior work exhibits very inconsistent performance. WASH is more consistent and improves average performance by 15%.

resulting in a 15% average performance improvement on these most challenging applications. These results probably understate AMP benefits with optimized microarchitectures.

Our VM scheduler is just as effective in a multiprogrammed workload consisting of a complex multithreaded adversary scheduled by the OS. Our VM approach adjusts even when the OS is applying an independent scheduling algorithm. A sensitivity analysis of our core model shows that we need a good predictor, but not a perfect one. In summary,

1. We demonstrate the power of information available within the VM for scheduling threads on AMP hardware.
2. We present automatic, accurate, and low-overhead VM dynamic analysis that: (a) classifies parallelism, (b) predicts core capabilities, (c) prioritizes threads holding contended locks, and (d) monitors thread progress.
3. We exploit this information in the WASH scheduler to customize its optimization strategy to the workload, significantly improving over other approaches.
4. Our implementation is available from the Jikes RVM research archive.

2. Related Work and Motivating Results

Table 1 qualitatively compares our approach to prior work with respect to algorithmic features and target workload. As far as we are aware, our approach is the only one to automatically identify bottlenecks in software and to comprehensively optimize *critical path*, *core sensitivity*, *priorities* and *load balancing*. Next, we overview how related work addresses these individual concerns and then present a quantitative analysis that further motivates our approach.

Critical path Amdahl's law motivates accelerating the critical path by scheduling it on the fastest core [8, 10–13, 22]. However, no prior work *automatically* identifies and prioritizes the critical path in software as we do. For instance, Joao et al.[12] use programmer hints and hardware to priorize threads that hold locks. Du Bois et al. [8] identify and accelerate critical thread(s) by measuring its useful work and the number of waiting threads with new hardware, but do not integrate into a scheduler. Our software approach automatically optimizes more complex workloads.

Priorities Scheduling low priority tasks, such as VM helper threads, OS background tasks, and I/O tasks, on small cores improves energy efficiency [5, 17], but these systems do not schedule the application threads on AMP cores. No prior AMP scheduler integrates priorities with application scheduling, as we do here.

Core sensitivity Prior work chooses the appropriate cores for competing threads using a cost benefit analysis based on speedup, i.e., how quickly each thread retires instructions on a fast core relative to a slow core [3, 6, 7, 14, 20, 21]. Systems model and measure speedup. Direct measurement executes threads on each core type and uses IPC to choose among competing threads [3, 7, 14]. To avoid migrating only for profiling and to detect phase changes, systems train predictive models offline with features such as ILP, pipeline-stalls, cache misses, and miss latencies collected from performance counters [6, 7, 20, 21]. At execution time, the models prioritize threads to big cores based on their relative speedup [6]. We combine profiling and a predictive model.

Load balancing *Static* schedulers do not migrate threads after choosing a core [14, 20], while *dynamic* schedulers adapt to thread behaviors [3, 7, 21] and load imbalance [15]. Li et al. schedule threads on fast cores first and ensure load is proportional to core capabilities, but do not consider core sensitivity nor complex workloads [15]. Saez et al. [21] and Craeynest et al. [7] both perform *proportional core-sensitive fair scheduling* (PFS) on scalable applications. The OS scheduler load balances by migrating threads between core types based on progress, thread sensitivity, and core capabilities. They simplify the problem by assuming the number of threads never exceed the number of cores: $|\text{threads}| \leq |\text{cores}|$ (pg. 20 [21]). Craeynest et al. compare to a scheduler that binds threads to cores for their entire execution, a poor baseline. Our approach dynamically adapts to thread behavior and handles $|\text{threads}| > |\text{cores}|$.

Quantitative Analysis We first examine the performance of PFS [7, 21], the best software only approach and compare it to the default round-robin Linux (Oblivious) scheduler, which seeks to keep all cores busy, and avoids thread migration [16, 18], and to Cao et al. [5] (bindVM), which simply binds VM helper threads to small cores. We execute 14 DaCapo Java benchmarks [4]. (Section 7 describes methodology in detail.) Figure 2 shows the performance on a one big / five small AMP configuration organized by workload type: sequential (ST), non-scalable multi-threaded (non-scalable MT), and scalable multi-threaded (scalable MT). We can see that (a) no one approach dominates across workload type; (b) bindVM performs best on sequential and non-scalable; (c) Oblivious and PFS perform essentially the same on scalable workloads, because in these workloads, threads out-number

Table 1: Qualitative comparison of WASH to related work.

	Algorithmic Features					Workload			
Approach	core sens.	load bal.	critical path	priorities	needs hints	limited to $\|T\| \leq \|C\|$	seq.	scale	non-scale
Becchi et al.[3] [3]	✓				no	no	✓		
Kumar et al. [14]	✓				no	yes	✓		
Craeynest et al. [7]	✓	✓			no	yes	✓	✓	✓
Saez et al. [21]	✓	✓			no	yes	✓	✓	
Du Bois et al. [8]			✓		yes	yes		✓	
Suleman et al. [22]			✓		yes	yes		✓	✓
Joao et al. [12]	✓		✓		yes	yes	✓	✓	✓
Li et al. [15]		✓			no	no	✓	✓	✓
Cao et al. [5]	limited			✓	no	no	✓	✓	✓
VM + WASH	✓	✓	✓	✓	***no***	***no***	✓	✓	✓

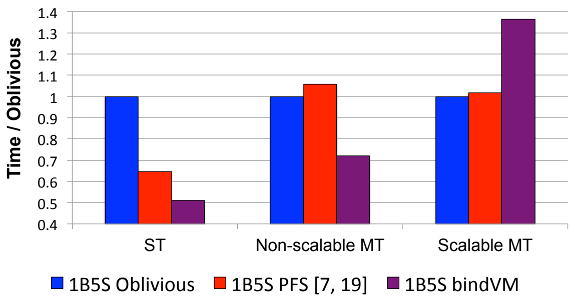


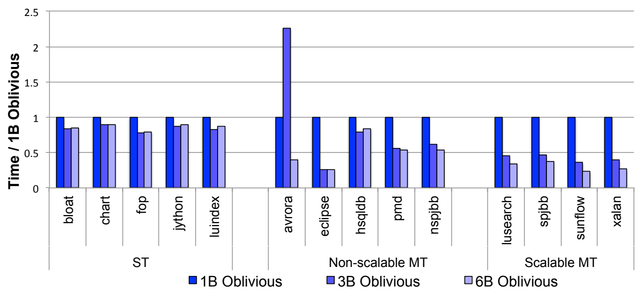
Figure 2: Quantitative time comparison of Linux (Oblivious), PFS, and bindVM on one big / five small (1B5S) AMP normalized to Oblivious. Lower is better. No approach dominates.

cores, and thus both reduce to round-robin scheduling across all cores. Since Oblivious and bindVM dominate PFS, we use them as our baseline throughout the paper. While bindVM is best on non-scalable workloads, WASH is better.

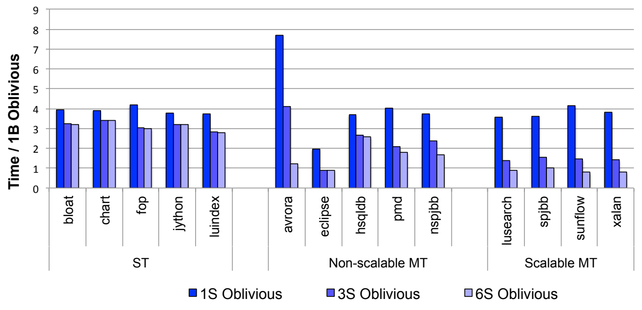
3. Workload Analysis

This section shows how to classify workloads based on their response to more cores. We explore scalability on small numbers of homogeneous cores by configuring a 6-core Phenom II to execute with 1, 3, and 6 cores at two speeds: big (B: 2.8 GHz) and small (S: 0.8 GHz). It then examines the strengths and weakness of Oblivious and bindVM as a function of workload type to motivate our approach.

Workload characteristics Figure 3(a) shows workload execution time on a big homogeneous multicore configuration and Figure 3(b) shows a small homogeneous multicore configuration, both normalized to Oblivious on one big core. Lower is better. Based on these results, we classify four of nine multithreaded benchmarks (lusearch, sunflow, spjbb and xalan) as scalable because they improve both from 1 to 3 cores, and from 3 to 6 cores. Five other multithreaded benchmarks respond well to additional cores, but do not improve consistently. For instance, avrora performs worse on 3 big cores than on 1; eclipse performs the same on 3 and 6 cores; and pjbb2005 does not scale in its default configuration. We create a scalable version of pjbb2005 by increasing its workload (spjbb). The original pjbb2005 is labeled nspjbb. The number of application threads and these results yield our sin-



(a) Time on one, three and six 2.8 GHz *big* cores.



(b) Time on one, three and six 0.8 GHz *small* cores.

Figure 3: Response to homogeneous parallelism with Linux (Oblivious) normalized to one big core where lower is better results in three workload types: single threaded, non-scalable multithreaded (MT), and scalable MT.

gle threaded, non-scalable multithreaded and scalable multithreaded categories.

Note single threaded applications in Figure 3 improve slightly as a function of core count. Because managed runtimes include VM helper threads, such as garbage collection, compilation, profiling, and scheduling, the VM process itself is multithreaded. Observing speedup as a function of cores in the OS *cannot* differentiate single-threaded from multithreaded applications in managed workloads. For example, fop and hsqldb have similar responses to the number of cores, but fop is single threaded and hsqldb is multithreaded.

AMP scheduling insights Consider again the results in Figure 2. Cao et al. previously showed that bindVM improves over Oblivious on one big and five small (1B5S) and the figure confirms this result. Regardless of the hardware configuration (1B5S, 2B4S, or 3B3S), bindVM performs best for single-threaded benchmarks because VM threads are non-critical and execute concurrently with the application thread. For performance, the application threads should always have priority over VM threads on big cores.

On scalable applications, Oblivious performs best and much better on 1B5S because bindVM restricts the application to the one big core, leaving small cores underutilized. On this Phenom II, one big core and five small cores has 41.6% more instruction execution capacity than six small cores. Ideally, a scalable parallel program will see this improvement on 1B5S. For scalable benchmarks, exchanging one small core for a big core with Oblivious boosts performance by 33%, short of the ideal 41.6%.

For non-scalable MT workloads, bindVM performs best on 1B5S, but improvements on 2B4S and 3B3S are limited. Intuitively, with only one big core, binding the VM threads gives application threads more access to the big core. With more big cores, Oblivious does a better job of load balancing. *Each scheduler performs well for some workloads, but no scheduler is best on all workloads.*

4. Dynamic Bottleneck Analysis

This section describes a new dynamic analysis that automatically classifies application parallelism and prioritizes bottleneck threads that hold locks. The VM periodically performs this analysis based on a configurable scheduling quantum. It is straightforward for the VM to count application threads separately from those it creates for GC, compilation, profiling, and other VM services. We add to the VM additional analysis of multithreaded applications that classify them as scalable or non-scalable by exploiting the Java Language Specification and lock implementation.

To identify threads that hold contended locks and accelerate them, we modify the VM to compute the ratio between the time each thread contends (waits) for another thread to release a lock and the total execution time of the thread thus far. When this ratio is high and the thread is responsible for a threshold of execution time as a function of the total available hardware resources (e.g., 1% with 2 cores, 0.5% with 4 cores, and so on), we categorize the benchmark as non-scalable. We set this threshold based on the number of cores and threads. The highest priority bottleneck thread is the lock-holding thread that is delaying the most work.

To prioritize among threads that hold locks, the VM piggybacks on the lock implementation and thread scheduler. When a thread tries to acquire a lock and fails, the VM scheduler puts the thread on a wait queue, a heavyweight operation. We time how long the thread sits on the wait queue using the RDTSC instruction, which incurs an overhead of around 50 cycles each call. At each scheduling quanta, the VM computes the waiting time for each thread waiting on a lock, then sums them, and then prioritizes the thread(s) that make other threads wait the longest to big cores. The VM implements biased locking, which lowers locking overheads by 'biasing' each lock to an owner thread, making the common case of taking an owned lock very cheap, at the expense of more overhead in the less common case where an unowned lock is taken [2]. Many lock implementations are similar. We place our instrumentation on this less frequent code path of a contended lock, resulting in negligible overhead.

Our critical thread analysis is more general than prior work because it automatically identifies bottlenecks in multithreaded applications with many ready threads and low priority VM threads, versus requiring new hardware [6] or developer annotations [11, 12]. Our analysis may be adopted in any system that uses biased locking or a similar optimiza-

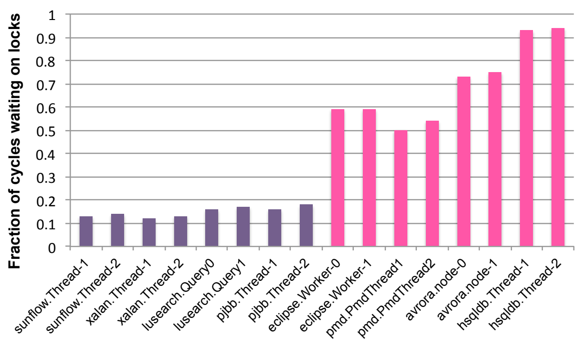


Figure 4: Fraction of time spent waiting on locks / cycles, per thread in multithreaded benchmarks. Left benchmarks (purple) are scalable, right (pink) are not. Low ratios are highly predictive of scalability.

tion. Modern JVMs such as Jikes RVM and HotSpot already implement it. Although by default Windows OS and Linux do not implement biased locking, it is in principle possible. For example, Android implements biased locking in its Bionic implementation of the pthread library [1, 2].

Figure 4 shows the results for representative threads from the multithreaded benchmarks executing on the 1B5S configuration. Threads in the scalable benchmarks all have low locking ratios and those in the non-scalable benchmarks all have high ratios. A low locking ratio is necessary but not sufficient. Scalable benchmarks typically employ homogeneous threads (or sets of threads) that perform about the same amount and type of work. When we examine the execution time of each thread in these benchmarks, their predicted sensitivity, and retired instructions, we observe that for spjbb, sunflow, xalan, and lusearch threads are homogeneous. Our dynamic analysis inexpensively observes the progress of threads, scaled by their core assignment, and determines whether they are homogeneous or not.

5. Speedup and Progress Prediction

To effectively schedule AMPs, the system must consider sensitivity to core features. When multiple threads compete for cores, the scheduler should execute the threads on big cores that benefit the most. For example, memory bound threads benefit little from a higher instruction issue rate.

Similar to prior work [7, 21], we create a predictive model offline that we use at run time. The model takes as input performance counters and predicts slow down and speedup, as appropriate. We use linear regression and Principal Component Analysis (PCA) to learn the most significant performance monitoring events and their weights. Since each processor may use only a limited number of performance counters, PCA analysis selects the most predictive ones.

Predicting speedup from little to big when the microarchitectures differ is often not possible. For example, with a single issue little core and a multiway issue big core, if the single issue core is always stalled, it is easy to predict that the thread will not benefit from more issue slots. However, if the single issue core is operating at its peak issue rate, no performance counter on it will reveal how much potential speedup will come from multi-issue. With a frequency-scaled AMPs, our model can and does predict both speedups and slow downs because the microarchitecture does not change.

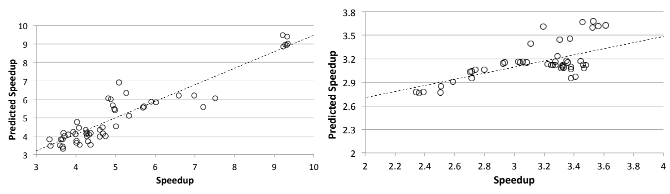
Performance counters

Intel	AMD
A: INSTRUCTIONS_RETIRED	X: RETIRED_INSTRUCTIONS
B: UNHALTED_REFERENCE_CYCLES	Y: RETIRED_UOPS
C: UNHALTED_CORE_CYCLES	Z: CPU_CLK_UNHALTED
D: UOPS_RETIRED:STALL_CYCLES	W: REQUESTS_TO_L2:ALL
E: L1D_ALL_REF:ANY	
F: L2_RQSTS:REFERENCES	
G: UOPS_RETIRED:ACTIVE_CYCLES	

linear prediction models

Intel	AMD
(-608B+609C+D+17E+27F-14G)/A	1.49+(1.87Y-1.08Z+27.08W)/X

Figure 5: PCA selects best performance counters to predict core sensitivity of threads with linear regression.



(a) Intel: i7 vs Atom **(b)** Phenom II: 2.8 vs 0.8 GHz.

Figure 6: Accurate prediction of thread core sensitivity. Y-axis is predicted. X-axis is actual speedup.

We explore the generality of our methodology and models using the frequency-scaled Phenom II and a *hypothetical* big/little design composed of an Intel Atom and i7. We execute and measure all of the threads with the comprehensive set of the performance monitoring events, including the energy performance counter on Sandy Bridge. We only train on threads that contribute more than 1% of total execution time, to produce a model on threads that perform significant amounts of application and VM work. We use PCA to compute a weight for each component (performance event) on a big core to learn the relative performance on a small core. We incrementally eliminate performance events with the same weights (redundancy) and those with low weights (not predictive), to derive the N most predictive events, where N is the number of simultaneously available hardware performance counters. We set N to the maximum that the architecture will report at once, four on the AMD Phenom II and seven on the Intel Sandy Bridge. This analysis results in the performance events listed in Figure 5. Figure 5 also shows a sample linear model for each architecture.

Figures 6(a) and 6(b) show the predictive power of the models for relative performance of the Intel processors and the frequency-scaled AMD Phenom II. We predict each application thread from a model trained on all the other applications threads, using leave-one-out validation. When executing a benchmark, we use the model trained on the other benchmarks in *all* experiments. The results show good predictive power of the PCA analysis.

Progress monitoring Our dynamic analysis monitors thread criticality and progress. It uses the retired instructions performance counter and scales it by the executing core capacity. Like many adaptive optimization systems, we predict that a thread will execute for the same fraction of time in the future as it has in the past. To correct for different core speeds, we normalize retired instructions based on the speedup prediction we calculate from the performance events. This normalization gives threads executing on small cores an opportunity to out-rank threads that execute on the big cores. Our model predicts fast-to-slow well for the i7 and Atom (Figure 6(a)). Our model predicts both slow-to-fast and fast-to-slow with frequency scaled AMD cores (Figure 6(b)). Thread criticality is decided based on predicted gain if it stays on or migrates to a big core. We present a sensitivity analysis to the model accuracy in Section 8.4.

6. The WASH Scheduling Algorithm

This section describes how we use core sensitivity, thread criticality, and workload to schedule threads when application and runtime threads exhibit varying degrees of heterogeneity and parallelism. We implement the WASH algorithm by setting thread affinities with the standard POSIX interface, which directs the OS to bind thread execution to one or more nominated cores. The VM assesses thread progress periodically (a 40 ms quantum) by sampling per-thread performance counter data and adjusts core affinity as needed. We require no changes to the OS. Because the VM monitors the threads and adjusts the schedule accordingly, even when the OS shares VM-scheduled cores with other competing complex multiprogrammed workloads (see Section 8.6), the VM scheduler adapts to OS scheduling choices.

Overview The scheduler starts with a default policy that assigns application threads to big cores and VM threads to small cores when they are created, following prior work [5]. For long-lived application threads, the starting point is immaterial. For very short lived application threads that do not last a full time quantum, this fallback policy accelerates them. All subsequent scheduling decisions are made periodically based on dynamic information. Every time quantum, WASH assesses thread sensitivity, criticality, and progress and adjusts the threads' affinity to cores accordingly.

We add to the VM the parallelism classification and the core sensitivity models described in Sections 4 and 5. The core sensitivity model takes as input performance counters for each thread and predicts how much the big core benefits the thread. The dynamic parallelism classifier uses a threshold for the waiting time. It examines the number of threads and dynamic waiting time to classify applications as single threaded, scalable, or non-scalable.

The VM stores a log of the execution history for each thread using performance counters and uses it: (1) to detect resource contention among application threads by comparing expected progress of each thread on its assigned core with its actual progress; and (2) to ensure that application threads have equal access to the big cores when there exist more ready application threads than big cores.

Algorithm 1 WASH

```
1:  function WASH($T_A$,$T_V$,$C_B$,$C_S$,$t$)
2:    $T_A$: Set of application threads
3:    $T_V$: Set of VM services threads, $T_A \cap T_V = \emptyset$
4:    $C_B$: Set of big cores
5:    $C_S$: Set of small cores, $C_B \cap C_S = \emptyset$
6:    $t$: Thread to schedule, where $t \in T_A \cup T_V$
7:    if $|T_A| \leq |C_B|$ then
8:      if $t \in T_A$ then Set Affinity of $t$ to $C_B$ return
9:      else Set Affinity of $t$ to $C_B \cup C_S$ return
10:   else if $t \in T_A$ then
11:     if $\forall \tau \in T_A(\text{Lock\%}(\tau) < \text{Lock}_{\text{Thresh}})$ then
12:       Set Affinity of $t$ to $C_B \cup C_S$ return
13:     else
14:       $T_{\text{Active}} \leftarrow \{\tau \in T_A : \text{IsActive}(\tau)\}$
15:       $T_{\text{Contd}} \leftarrow \{\tau \in T_A : \text{Lock\%}(\tau) > \text{Lock}_{\text{Thresh}}\}$
16:       if $ExecRank(t, T_{\text{Active}}) < ER_{\text{Thresh}}|C_B|$ or
17:           $LockRank(t, T_{\text{Contd}}) < LR_{\text{Thresh}}|C_B|$ then
18:         Set Affinity of $t$ to $C_B$ return
19:       else Set Affinity of $t$ to $C_B \cup C_S$ return
20:   else Set Affinity of $t$ to $C_S$ return
```

Algorithm 1 shows the pseudo-code for the WASH scheduling algorithm. WASH makes three main decisions. (1) When the number of application threads is less than or equal to the number of big cores, WASH schedules them on the big cores (line 8). (2) WASH classifies a workload as scalable when no thread is a bottleneck and there is no contention. For simplicity, the pseudo-code omits the code that detects contention using thread progress from historical information. For scalable workloads, the algorithm gives all threads equal access to the big cores (lines 11-12). This strategy follows the default round robin OS scheduler. If we detect contention (not shown), we schedule the application threads as if the application is non-scalable. (3) For non-scalable workloads (line 14-19), the algorithm prioritizes application threads whose instruction retirement rates on big cores match the rate at which the big cores can retire instructions weighted by its historical access to big cores, computing an *execution rank* (line 16). WASH also uses the accrued lock waiting time to prioritize the bottleneck threads on which other threads wait the most, computing a *lock rank* (line 17). WASH prioritizes a thread to the big cores in both cases. VM service threads are scheduled to the small cores (line 20), unless the number of application threads is less than the number of big cores. The next 3 subsections discuss each case in more detail.

Single-Threaded and Low Parallelism When the application creates a thread, WASH's fallback policy sets the thread's affinity to big cores. At each time quantum, WASH assesses the thread schedule. When WASH dynamically detects one application thread or the number of application threads $|T_A|$ is less than or equal to the number of big cores $|C_B|$, then WASH sets the affinity for the application threads to $|T_A|$ of the big cores (line 8). WASH also sets the affinity for the VM threads such that they may execute on the remaining big cores or on the small cores. It sets the affinity of VM threads to all cores (line 9), which translates to the relative complement of $C_B \cup C_S$ with respect to $|T_A|$ big cores being used by T_A. If there are no available big cores, WASH sets the affinity for all VM threads to execute on the small cores, following Cao et al. [5]. Single-threaded applications are the most common example of this scenario.

Scalable multithreaded When the VM detects a scalable $|T_A| > |C_B|$ and homogenous workload, then the analysis in Section 3 shows that Linux's default CFS scheduler does an excellent job of scheduling application threads. We use our efficient lock contention analysis to establish scalability (line 11). We empirically established a threshold of 0.5 contention level (time spent contended / time executing) beyond which a thread is classified as contended (see Figure 4). WASH monitors the execution schedule from the OS scheduler and ensures that all of the homogeneous application threads make similar progress (not shown in the pseudocode). If any thread falls behind, for example, by spending a lot of time on a small core or because the OS has other competing threads to schedule, WASH boosts its priority and binds it to a big core. It thus reprioritizes the threads based on their expected progress. In this case, WASH treats the application as non-scalable and schedules it accordingly, as described below.

Non-scalable multithreaded WASH The most challenging case is how to prioritize non-scalable application threads when the number of threads outstrips the number of cores and all threads compete for both big and small cores. Our algorithm is based on two main considerations: (a) how critical the thread is to the overall progress of the application (lock information), and (b) the relative capacity of big cores compared to small cores to retire instructions for each thread.

We rank threads based on their relative capacity to retire instructions, seeking to accelerate threads that dominate in terms of productive work (line 16). For each active thread (non-blocked for the last two scheduling quantum), we compute *ExecRank*: a rank based on the running total of retired instructions, corrected for core capability and their access to big and small cores in previous scheduling quantas. If a thread runs on a big core, we accumulate its retired instructions from the dynamic performance counter. When the thread runs on a small core, we increase its retired instructions by multipling it by predicted speedup from executing on the big core. Thus we assess importance on a level playing field — judging each thread's progress as if it had access to large cores. Then, we compose this amount with the predicted speedup for all threads. Notice that threads that will benefit little from the big core will naturally have less importance (regardless of which core they are running on in any particular time quantum), and that conversely threads that will benefit greatly from the big core will have be ranked accordingly. We call this rank computation *adjusted priority* and compute it for all active threads. We rank all active threads based on this adjusted priority. We also compute a *LockRank*, which prioritizes bottlenecks based on the amount of time other threads have been waiting for it (line 17).

We use the relative rank to select a set threads to execute on the big cores. Note, rank 1 is the top ranked most important thread and $|T_A|$ is the lowest. We do *not* size the set according to the fraction of cores that are big ($B/(B+S)$), but *instead* size the thread set according to the big cores' relative capacity to retire instructions ($B_{RI}/(B_{RI}+S_{RI})$). For example, in a system with one big core and five small cores, where the big core can retire instructions at five times the rate of each of the small cores, the size of the set would be $B_{RI}/(B_{RI}+S_{RI}) = 0.5$. In that case, we will assign to the big cores the 1 to N most important threads such that the adjusted retired instructions rate of those N threads is 0.5 of the total in this example (line 16). We also select a set of the highest lock-ranked threads to execute on the big cores. We size this set according to the fraction of cores that are big ($B/(B+S)$) (line 17).

The effect of this algorithm is twofold. First, overall progress is maximized by placing the threads that are both critical to the application and that will benefit from the speedup on the big cores. Second, we avoid over or under subscribing the big cores by scheduling according to the relative capacity of those cores to retire instructions. VM helper threads actually benefit from the big core in some cases [5] (see Section 8.5), but WASH ensures application threads get priority over them. Furthermore, WASH explicitly focuses on non-scalable parallelism. By detecting contention and modeling total thread progress (regardless of core assignment), our model corrects itself when threads compete for big cores yet cannot get them.

Summary The WASH application scheduler customizes its strategy to the workload, applying targeted heuristics, accelerating the critical path, prioritizing application threads over low-priority helper threads to fast cores, and proportionally scheduling parallelism among cores with different capabilities and effectiveness based on the thread progress.

7. Methodology

We measure and report performance, power, and energy, leveraging prior tools and methodology [4, 5, 9]. Our system is publicly available in the Jikes RVM research repository.

Hardware configuration methodology We measure and report performance, power, and energy on the AMD Phenom II because it supports independently clocking each core with DVFS. Prior work establishes this methodology for evaluating AMP hardware [5, 21]. Concurrently with this work, Qualcomm announced AMP Snapdragon hardware with 4 big and 4 little cores, but it was not available when we started and it uses the ARM ISA, whereas our tools target x86. Compared to simulation, there are no accuracy questions, but we explore fewer hardware configurations. We measure

Table 2: Experimental processors.

	i7	**Atom**	**Phenom II**
Processor	Core i7-2600	AtomD510	X6 1055T
Architecture	Sandy Bridge	Bonnell	Thuban
Technology	32 nm	45 nm	45 nm
CMP & SMT	4C2T	2C2T	6C1T
LLC	8 MB	1 MB	6 MB
Frequency	3.4 GHz	1.66 GHz	2.8 GHz & 0.8 GHz
Transistor No	995 M	176 M	904 M
TDP	95 W	13 W	125 W
DRAM Model	DDR3-1333	DDR2-800	DDR3-1333

existing hardware orders of magnitude faster than simulation and consequently explore more software configurations.

Table 2 lists characteristics of the machines we use. (We only use the Sandy Bridge and Atom to show the core prediction model generalizes in Section 4). We statically configure the Phenom II as an AMP by setting each core's voltage and frequency at boot time. To measure power and energy, we use Cao et al.'s Hall effect sensor methodology [5, 9]. All the performance, power, and energy results in this section use the Phenom II.

Operating System We use Ubuntu 12.04 with the default 3.8.0 Linux kernel. The default Linux CFS scheduler is oblivious to different core capabilities, seeks to keep all cores busy and balanced based on the task numbers on each core, and tries not to migrate threads between cores [16, 18].

Virtual machine configuration We add our analysis and scheduler to Jikes RVM [23]. The VM scheduler executes periodically. We choose a 40 ms time quantum following prior work [6], which shows no discernible thread migration overhead on shared-LLC AMP processors with 25 ms. All measurements follow Blackburn et al.'s best practices for Java performance analysis with the following modifications of note [4]. We measure first iteration, since we explore scheduling JIT threads. We use concurrent Mark-Sweep collection, and collect every 8 MB of allocation for avrora, fop and luindex, which have the highest rates of allocation, and 128 MB for the others. We configure the number of collection threads to be the same as available cores. We use default JIT settings in Jikes RVM, which intermixes compilation with execution. Jikes RVM does not interpret. A baseline compiler JITs code upon first execution and then the compiler optimizes at higher levels when its cost model predicts the optimized code will amortize compilation costs.

Workload We use thirteen Java benchmarks taken from DaCapo: bloat, eclipse, fop, chart, jython and hsqldb (DaCapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (DaCapo-9.12); and from SPEC, pjbb2005 [4]. We use all the DaCapo benchmarks that execute on the unmodified Jikes RVM. These benchmarks are non-trivial real-world open source Java programs under active development [4]. Finding that pjbb2005 does not scale well, we create a second scalable version by increasing the number of transactions from 10,000 to 100,000, yielding spjbb, which scales on our six core machine. We use the workload classification from Section 3 to organize the results and analysis.

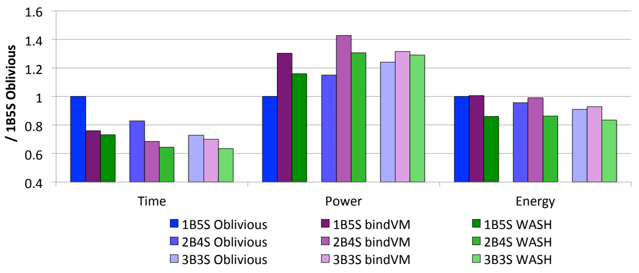


Figure 7: Geomean time, power and energy with Oblivious, bindVM, and WASH on all three hardware configs.

Comparisons We compare WASH to two baselines: the default OS scheduler (Oblivious) with no VM direction and bindVM [5], which simply binds VM helper threads to the small cores using the OS set affinity interface. We use the unmodified bindVM implementation from the Jikes VM repository. These schedulers are the only ones that handle general workloads automatically, e.g., messy non-scalable MT workloads with a mix of application and VM threads, no programmer intervention to identify bottleneck locks, and/or no new hardware. Section 8.5 shows that an approximation of the closest prior work [7, 21] performs much worse than WASH, Linux, and bindVM because it does not consider thread priorities (VM versus application threads) nor prioritize threads that create bottlenecks by holding locks, contributions of our work.

Measurement Methodology We execute each configuration 20 or more times, reporting first iteration results, which mix compilation and execution. We omit confidence intervals from graphs. For the WASH scheduling algorithm, the largest 95% confidence interval for time measurements with 20 invocations is 5.22% and the average is 1.7%. For bindVM, the largest 95% confidence interval for time measurements with 20 invocations is 1.64% and the average is 0.72%. Oblivious has the largest 95% confidence interval; with 20 invocations, the largest interval is 15% and the average is 5.4%. Thus, we run the benchmarks with Oblivious for 60 invocations, lowering the largest error to 9.6% and the average error to 3.7%. The relatively high confidence intervals result because performance is sensitive to the thread core mapping and all the systems have non-determinism due to dynamic triggers for scheduling, garbage collection, and JIT compilation. The 95% confidence intervals are a good indicator of performance predictability of the algorithms.

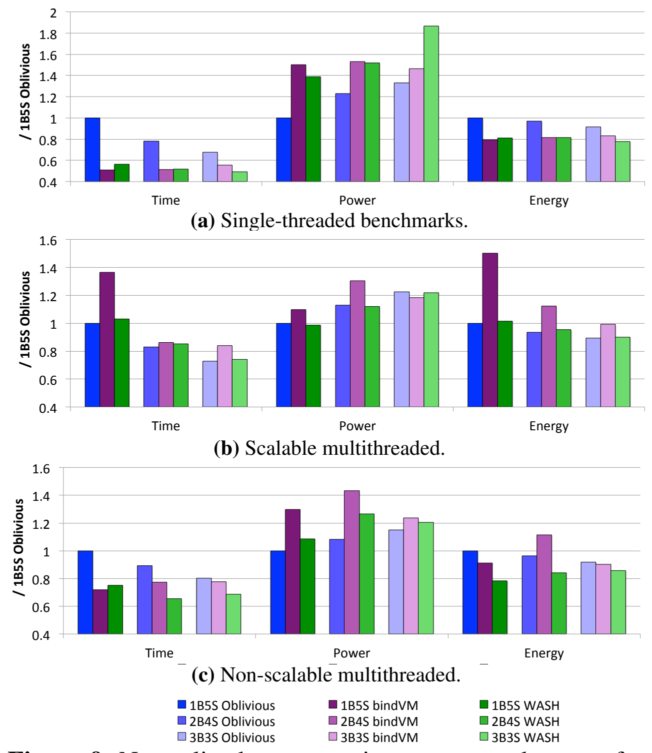
8. Results

Figure 7 summarizes the performance, power, and energy results on three AMD hardware configurations: 1B5S (1 Big core and 5 Small cores), 2B4S and 3B3S. We weight each benchmark group (single threaded, scalable, and non-scalable) equally. Figure 8 shows all the individual benchmark results on the same three hardware configurations. We normalize to Oblivious, lower is better.

Figure 7 shows that WASH improves performance and energy on all three hardware configurations on average. Oblivious has the worst average time on all of the configurations and even though it has the lowest power cost, up to 16% less power than WASH, it still consumes the most energy. Oblivious treats all the cores the same and evenly distributed threads, with the result that the big core may be underutilized and critical threads may execute unnecessarily on a small core.

WASH attains its performance improvement by using more power than Oblivious, but at less additional power than bindVM. The bindVM scheduler has lower average time compared to Oblivious, but it has the worst energy and power cost in all hardware settings, especially on 2B4S. bindVM uses up to 18% more energy than WASH. The bindVM scheduler overloads big cores with work that can be more efficiently performed by small cores, leading to higher power and underutilization of small cores.

WASH and bindVM are especially effective compared to Oblivious on 1B5S, as the importance of correct scheduling decisions is most exposed. On 1B5S, WASH reduces the geomean time by 27% compared to Oblivious, and by about 5% comparing to bindVM. For energy, WASH saves more than 14% compared to bindVM and Oblivious. WASH consistently improves over bindVM on power. The following subsections structure a detailed analysis based on workload categories.



(a) Single-threaded benchmarks.

(b) Scalable multithreaded.

(c) Non-scalable multithreaded.

Figure 9: Normalized geomean time, power and energy for different benchmark groups. Lower is better.

8.1 Single-Threaded Benchmarks

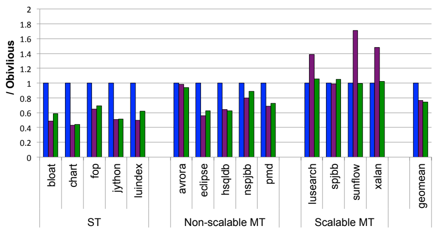
Figure 9a) shows that for single-threaded benchmarks, WASH performs very well in terms of total time and energy on all hardware settings, while Oblivious performs poorly. WASH consumes the least energy on all hardware settings. Compared to Oblivious scheduling, WASH reduces execution time by as much as 44%. WASH lowers energy by 19% but increases power by as much as 39% compared to Oblivious. Oblivious performs poorly because it is unaware of the heterogeneity among the cores, so with high probability in the 1B5S case, it schedules the one application thread onto a small core. With only one application thread and one big core, both WASH and bindVM will schedule the the thread to the big core and VM threads to the small cores. When the number of big cores increases, there is a smaller distinction between the two policies because the VM threads may be scheduled on big as well as small cores. In steady state the other VM threads do not contribute greatly to total time, as long as they do not interfere with the application thread. Note that power consumption is higher for bindVM and WASH than for Oblivious. When the one application thread migrates to the small cores, it consumes less power compared to bindVM and WASH, but the loss in performance more than outweighs the decrease in power. Thus total energy is reduced by WASH. In the single-threaded scenario, WASH and bindVM perform very similar to each other on all AMP configurations.

8.2 Scalable Multithreaded Benchmarks

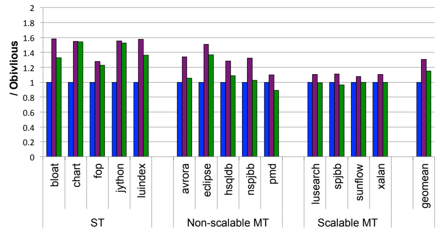
Figure 9b) shows that for scalable MT benchmarks, WASH and Oblivious perform very well in both execution time and energy on all hardware configurations, while bindVM performs poorly. Compared to WASH and Oblivious scheduling, bindVM increases time by as much as 36%, increases energy by as much as 50%, and power by as much as 15%. The reason bindVM performs poorly is that it simply binds all application threads to the big cores, leaving the small cores under-utilized. Unsurprisingly, as the number of small cores decreases and the number of large cores increases, the difference between bindVM compared to Oblivious and WASH reduces.

Scalable benchmarks with homogeneous threads benefit from round-robin scheduling policies as long as the scheduler migrates threads among the fast and slow cores frequently enough. When threads out number cores, as they often do in multithreaded managed workloads, Oblivious is forced to migrate threads. Therefore even though Oblivious does not reason explicitly about the relative core speeds, it migrates threads frequently enough, achieving very good performance, power, and energy.

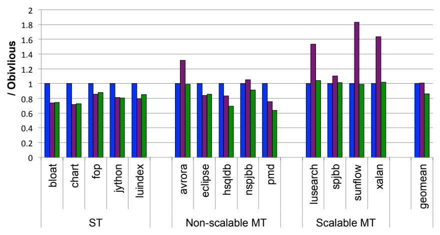
By using the contention information it gathers online, WASH correctly identifies scalable benchmarks. WASH and Oblivious generate similar results, but through different means. WASH reasons explicitly about relative core speeds using historical execution data to migrate threads propor-



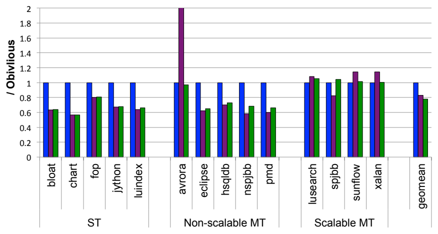
(a) Time on 1B5S.



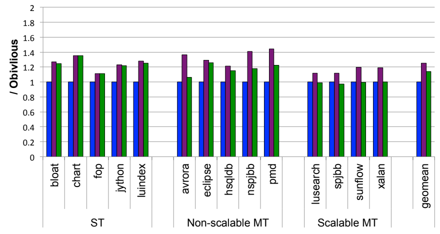
(b) Power on 1B5S.



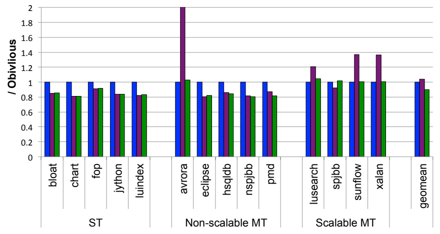
(c) Energy on 1B5S.



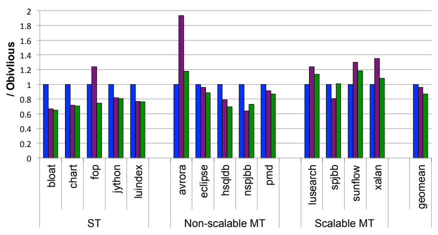
(d) Time on 2B4S.



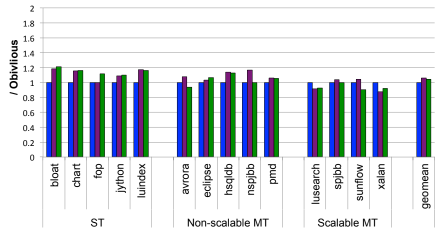
(e) Power on 2B4S.



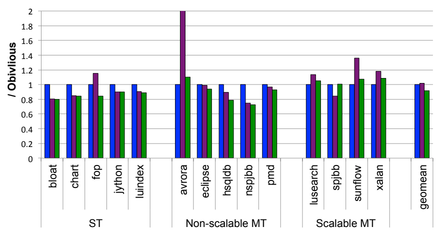
(f) Energy on 2B4S.



(g) Time on 3B3S.



(h) Power on 3B3S.



(i) Energy on 3B3S.

🟦 Oblivious 🟪 bindVM 🟩 WASH

Figure 8: Time, power, and energy for all benchmarks on all hardware configurations.

tionally between slow and fast cores. This analysis adds a small amount of overhead in some cases, but in others results in slightly better scheduling of application threads to small cores. On average, WASH and Oblivious both achieve similar good results on scalable multithreaded applications.

8.3 Non-scalable Multithreaded Benchmarks

Figure 9c) shows that WASH on average performs best and neither Oblivious or bindVM is consistently second best in the more complex setting of non-scalable MT benchmarks. For example, eclipse has about 20 application threads and hsqldb has about 80. They each have high degrees of contention. In eclipse, the *Javaindexing* thread consumes 56% of all of the threads' cycles while the three *Worker* threads consume just 1%. In avrora threads spend around 60-70% of their cycles waiting on contended locks, while pmd threads spend around 40-60% of their cycles waiting on locks. These *messy* workloads make scheduling challenging.

For eclipse and hsqldb in Figure 8, the results for WASH and bindVM are similar with respect to time, power, and energy in most hardware settings. The reason is that even though eclipse has about 20 and hsqldb has 80 threads, in both cases only one or two of them are dominant. In eclipse, the threads *Javaindexing* and *Main* are responsible for more than 80% of the application's cycles. In hsqldb, *Main* is responsible for 65% of the cycles. WASH will correctly place the dominant threads on big cores, since they have higher priority. Most other threads are very short lived, shorter than our 40 ms scheduling quantum. Since before the profile is gathered, WASH binds application threads to big cores, the short-lived threads will just stay on big cores. Since bindVM will put all application threads on big cores and for these two benchmarks, one thread dominates, the results for the two policies are similar.

WASH improves energy efficiency with similar performance compared to bindVM. The benchmarks avrora, nspjbb, and pmd in Figure 8 are good examples of WASH choosing to execute on a small core application threads that will not benefit as much from a big core. Particularly, for 1B5S, compared to WASH, bindVM time is lower; however, because WASH makes better use of the small cores, WASH decreases power usage compared to bindVM. Since bindVM does not reason about core sensitivity, it does not make a performance/power trade-off. WASH makes better use of small cores, improving average energy efficiency compared to bindVM for avrora, nspjbb, and pmd on all hardware configurations.

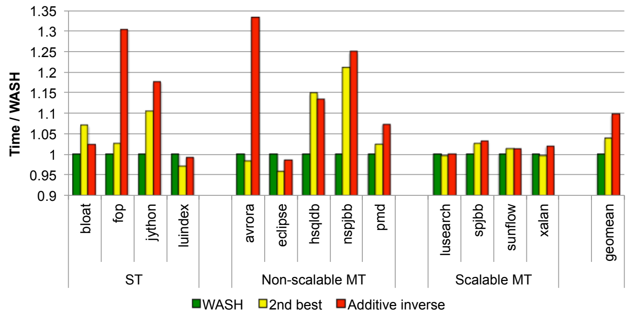


Figure 10: WASH with best (default) model, second best model, and a bad model (inverse weights) on 1B5S.

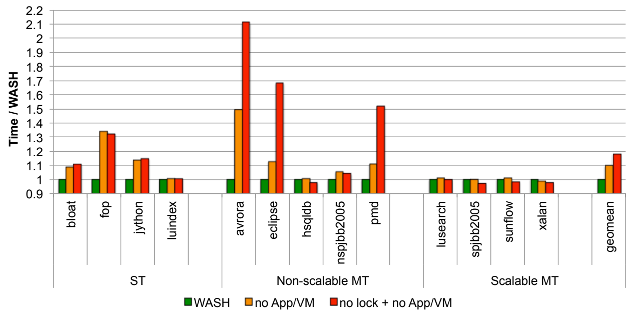


Figure 11: WASH significantly out-performs PFS [7, 21] which lack lock analysis and VM vs application priority on 1B5S.

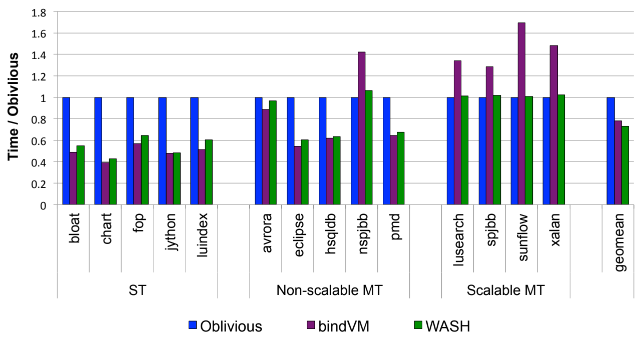


Figure 12: Performance with eclipse adversary on 1B5S.

8.4 Sensitivity to Speedup Model

Figure 10 shows the sensitivity of WASH to its speedup model with a) the default best model from Section 5, b) a model using the next best 4 different hardware performance counters from the same training set, and c) a model with the additive inverse weights of the default one. WASH only degrades by 2-3% when we change the model to slightly worse one. However, a bad model for the speedup prediction, degrades performance by 9%.

8.5 Thread & Bottleneck Prioritization

We build a version of Proportional core-sensitive Fair Scheduling (PFS) [7, 21], by removing features unique to WASH: a priori knowledge to preferentially schedule application threads to big cores and automatically prioritizing among contended locks. The resulting PFS scheduler performs proportional fair scheduling for homogeneous workloads on AMP and prioritizes threads based on core sensitivity. Figure 11 shows that as expected, these schedulers (red — no lock + no APP/VM) perform well on scalable benchmarks, consistent with prior results [7, 21] that test on workloads with total |threads| = |cores| and with parallelism that, for the most part, is homogeneous and scalable. However, the prior work performs poorly on sequential and non-scalable parallel benchmarks. The orange bars (no App/VM) show just disabling prioritization of application over VM threads, revealing that this feature is critical to good performance for managed single-threaded and non-scalable workloads. We find that the JIT thread benefits from big cores more than most application threads, so core sensitivity scheduling alone will mistakenly schedule it on the big core even though it is often not critical to application performance. WASH correctly deprioritizes JIT threads to small cores. Highly concurrent messy workloads, such as avrora and eclipse, suffer greatly when the scheduler does not prioritize contended locks. Avrora uses threads as an abstraction to model high degrees of concurrency for architectural simulation. Eclipse is a widely used Integrated Development Environment (IDE) with many threads that communicate asynchronously. Both are important examples of concurrency not explored, and consequently not addressed, by the prior work. WASH's comprehensive approach correctly prioritizes bottleneck threads in these programs, handling more general and challenging settings.

8.6 Multiprogrammed Workloads

This section presents an experiment with multiprogrammed workload in the system, of which our VM is unaware. We use eclipse as the OS scheduled adversarial workload across all cores and WASH scheduling on each of our benchmarks. Eclipse has 20 application threads with diverse needs and is demanding both computationally and in terms of memory consumption.

Figure 12 shows the performance of WASH and bindVM compared to Oblivious in the presence of the eclipse adversary. Although bindVM's overall performance is largely unchanged compared to execution with no adversary, it degrades on both nspjbb and spjbb. WASH performs very well in the face of the adversary, with average performance 27% better than and a number of benchmarks performing twice as well as Oblivious. WASH's worse case result (nspjbb) is only 7% worse.

Summary The results show that WASH improves performance and energy on average over all workloads, each component of its algorithm is necessary and effective, and it is robust to the introduction of a non-trivial adversary. WASH utilizes both small and big cores as appropriate to improve performance at higher power compared to Oblivious, which under-utilizes the big cores and over-utilizes little cores because it does not reason about them. On the other hand, WASH, uses its workload and core sensitivity analysis to improve performance and lower power compared to bindVM which under utilizes the little cores for the scalable and non-scalable MT benchmarks.

9. Conclusion

Hardware heterogeneity is a promising approach to improving performance and energy. However, only if this complexity is transparent to applications is software likely to benefit from it as hardware continues to evolve. This paper shows how to modify a VM to analyze bottlenecks, AMP sensitivity, and progress to deliver transparent, portable performance, and energy efficiency. We introduce new fully automatic dynamic analyses that identify scalable parallelism, non-scalable parallelism, bottleneck threads, and thread progress. We show that this system delivers substantial improvements in performance and energy efficiency on frequency-scaled processors over prior software approaches. Our results likely understate the advantages from more highly optimized commercial AMP systems that vendors are beginning to deliver.

Acknowledgments

This research is funded by the China Postdoctoral Science Foundation (No. 2015T80139), the National Natural Science Foundation of China (No. 61432018, 61272136, 61133005, 61221062), the National High Technology Research and Development Program of China (No. 2015AA01A303, 2015AA 011505), and the National Science Foundation of the United States (SHF-0910818).

References

[1] Android. Bionic platform, 2014. URL `https://github.com/android/platform_bionic`.

[2] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI'98*, pages 258–268, 1998.

[3] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Computing Frontiers*, pages 29–40, 2006. ISBN 1-59593-302-6.

[4] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, Oct. 2006.

[5] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA'12*, pages 225–236, 2012.

[6] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narváez, and J. S. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ISCA'12*, pages 213–224, 2012.

[7] K. V. Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. *PACT'13*, pages 177–187, 2013.

[8] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In *ISCA'13*, pages 511–522, Jun. 2013.

[9] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ASPLOS*, pages 319–332, 2011.

[10] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[11] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS 2012*, pages 223–234, 2012.

[12] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded application on asymmetric CMPs. In *ISCA'13*, pages 154–165, 2013.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[14] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA'04*, pages 64–75, 2004.

[15] T. Li, D. P. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC'07*, pages 1–11, 2007.

[16] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP'09*, pages 65–74, 2009.

[17] J. C. Mogul, J. Mudigonda, N. L. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *Micro*, 28(3):26–41, 2008.

[18] I. Molnar. Modular Scheduler Core and Completely Fair Scheduler [CFS]. `http://lwn.net/Articles/230501/`, Apr. 2007.

[19] Qualcomm. Snapdragon 810 processors, 2014. URL `https://www.qualcomm.com/products/snapdragon/processors/810`.

[20] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *JPDC*, 71(1):114–131, 2011.

[21] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM TCM*, 30(2):6:1–38, 2012.

[22] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[23] The Jikes RVM Research Group. Jikes Open-Source Research Virtual Machine, 2011. URL `http://www.jikesrvm.org`.