System Design for Heterogeneity: The Virtual Machine Services Test Case

Ting Cao*, Tiejun Gao*, Stephen M Blackburn*, and Kathryn S McKinley†
*Australian National University †Microsoft Research
† The University of Texas at Austin
{Ting.Cao, Tiejun.Gao, Steve.Blackburn}@anu.edu.au, mckinley@microsoft.com

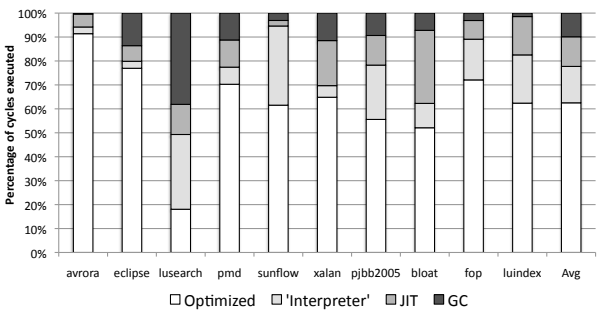
Abstract

Power and energy constraints are forcing architects to propose heterogeneous parallel systems. Unfortunately, tuning or customizing parallel hardware for a particular application compromises application portability. This paper focuses on the class of heterogeneous systems that combine general-purpose big and small cores for portability. This paper identifies key software characteristics and a framework for exploiting them to improve performance, power, and energy on big/small multicores: (1) parallelism, (2) asynchrony, (3) criticality, and (4) hardware sensitivity. We choose Virtual Machine (VM) services executing managed languages as our workload. They are pervasive (e.g., JavaScript, PHP, Java, and C#), dominate power and energy constrained platforms, such as the phone and tablet, do not compromise application portability, and use a lot of energy. VM services interpret, compile, profile, schedule, and manage memory while executing applications. Fortunately, VM services are often parallel, asynchronous, not on the application's critical path, and are power and energy efficient on small cores. This paper demonstrates (a) that VM services consume a lot of power and energy, almost 40% in modern systems; (b) how to exploit unique combinations of the four characteristics in VM services to lower power, improve performance, and improve performance per energy (PPE); and (c) that using big cores for application threads together with tuned small low-power cores for VM services can substantially lower power and improve performance per energy (PPE). These results suggest that other system services, additional architecture tuning or customization, and software/hardware co-design are fruitful future directions for improving PPE and meeting power constraints.

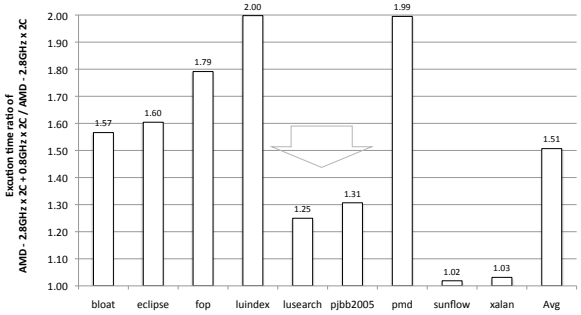
I. Introduction

Computer systems are facing a power crisis. Because Dennard scaling is reaching its limits, future systems will contain "dark silicon" [1], [2] — the fraction of on-chip transistors powered at once will decrease exponentially. For mobile devices, battery life always falls behind demand. For most of data centers, the cost of electricity is now the top budgetary consideration [3]. The EPA estimated that U.S. data center electricity consumption cost over $4.5 billion in 2006 and projected growth to $7.4 billion in 2011 [4]. Consequently, Google purchases computers with the best performance per energy (PPE) dollar, rather than the best absolute performance [5], and the Japanese Green IT Council promotes PPE as a world standard metric for data center efficiency [6].

Architects are turning to customization and heterogeneity to lower power and improve PPE [7], [8], [1], [9], [10], [11], [12]. Customized hardware for a specific function is well known to provide orders of magnitude improvements in performance, power, or both, but offers a daunting programming task and the resulting software is generally not portable. Consider GPUs. The GFLOPs per Watt of the AMD GPU 9270 is about six times higher than a general-purpose Core i7-920. Motivated expert programmers specialize to this hardware, but software portability is compromised, sometimes even to the next generation of the same GPU. To address the programming problem, researchers are exploring generating FPGAs from high-level language descriptions [13], but success will depend on driving down costs. This paper explores a middle-ground: single ISA heterogeneous multicore architectures, which combine general-purpose big and small multicores, where the big cores are tuned for high performance and the small cores are tuned for low power [14], and possibly tuned to specific loads. For example, ARM just announced the combination of its



(a) Fraction of cycles in VM services on i7 1C1T 3.4 GHz with a generational Immix GC. See Section III for methodology.



(b) Performance impact of *adding* two slow cores on an AMD Phenom II.

Fig. 1. Motivation: (a) VM services consume significant resources. (b) Naive addition of slow cores *slows* down applications.

highest-performing low-power Cortex-A15 and its most energy efficient Cortex-A7 [7]. Texas Instruments announced the OMAP 5 platform which includes two ARM A15 cores and two low power ARM M4 cores on a single 28nm die along with eight special purpose processors for graphics and video[15]. Intel built an experimental machine that combines its Nehalem and Atom architectures in a system on a chip [10], [11]. By assigning the critical path to the big core and other execution to the simple core researchers show how to improve performance and consequently energy [14], [10], [11], [12].

This paper identifies and leverages unique combinations of four software characteristics: (1) parallelism, (2) asynchrony, (3) criticality and (4) hardware sensitivity. It goes beyond improving performance on general-purpose heterogeneous big/small multicores to improve PPE and meet power constraints. Within this framework, this paper explores a test case: Virtual Machine (VM) services executing managed languages on big/small multicores. VM services are *an attractive choice* because they are ubiquitous and do not compromise application portability. Smart phone and tablet applications are exclusively written in managed languages. The vast majority of client and server side web applications use JavaScript and PHP. Business applications are increasingly written in Java and C#. Since VM services (e.g., interpreter, compiler, profiler, and garbage collector) execute together with every application, exploiting big/small multicores to improve power, performance, and PPE will transparently improve all managed applications.

This paper shows VM services are also *a lucrative choice* because they consume almost 40% of total time and energy and they exhibit the requisite software characteristics. Figure 1(a) shows the fraction of cycles Java applications spend in VM services. GC consumes 10% and JIT consumes 12% of all cycles on average. An additional 15% of total time is spent executing unoptimized code (e.g., via the interpreter). Total time in VM services ranges from 9% to 82%. Prior GC performance results on industry VMs confirm this trend: IBM's J9, JRockit, and Sun (now Oracle) HotSpot JDK actually show an even higher average fraction of time spent on garbage collection [16]. VM services in less mature VMs, such as JavaScript and PHP VMs, likely consume an even higher fraction of total cycles. Reducing power and PPE of VM services thus makes a promising target.

Figure 1(b) shows that simply executing managed applications on a big/small multicore platforms without any VM or OS support is a very bad idea. On the AMD Phenom II, we compare executing Java applications in Jikes RVM on two cores executing at their highest frequency to two cores at their highest frequency *plus* two cores at their lowest. Even though this big/small multicore configuration provides more hardware resources, it slows down applications by more than 50%! Using hardware and software configuration, and power measurements, we show how to exploit the unique characteristics of garbage collector, interpreter, and just-in-time optimizing compiler (JIT) workloads to improve total power, performance, energy, and/or PPE on big/small multicores.

Garbage Collection. Because garbage collection is memory bound, a parallel collector improves PPE with a tuned low-power core, whether or not the collector is asynchronous (characteristics 1, 4, and 2). Many high-performance, power-hungry hardware features that improve application PPE are inefficient for

GC (and interpreter): GC does not benefit from a high clock rate or instruction level parallelism (ILP), but it does benefit from coarse-grain parallelism and memory bandwidth.

JIT. We show that because a parallel JIT is asynchronous and not-critical (1-3), sacrificing its performance by executing it on small cores lowers power and improves performance, energy, and thus PPE. Modern high-performance architecture features executing at the highest frequency with the most hardware parallelism, bandwidth, and cache all increase the PPE of the applications and JIT. However, the JIT consumes much less power than the applications and running the JIT in a separate simple core improves application performance, energy, and PPE because it forces the JIT off the critical path and the JIT delivers optimized code fast enough at much lower power. We show that with this hardware, we can make the JIT more aggressive, such that it delivers better application code faster, with little power cost.

Interpreter. Executing the interpreter on small cores improves PPE because the interpreter's parallelism matches the application's. Even though the interpreter may contribute to the critical path, it is not asynchronous (1-3). The interpreter has low ILP, a small cache footprint, and does not use much memory bandwidth. These results indicate that executing an interpreter on a high performance CPUs is inefficient.

Each service offers a different combination of software characteristics, yet we show how to configure each to lower power, improve performance, improve PPE, or in some cases all three. The paper's contributions show (1) VM services are significant consumers of resources, (2) without hardware/software co-design big/small multicores are not effective, and (3) configuring VM services to exploit small cores will deliver substantially better total power and PPE. Since VM services' algorithms depend on the hardware, our results suggest additional software/hardware co-design should drive down PPE for managed languages further.

II. VM Background

Modern virtual machines (VM) services for managed languages include garbage collection (GC) for memory safety, dynamic interpretation and/or just-in-time (JIT) compilation for portability, and dynamic profiling and JIT optimizations for performance.

Garbage Collection. Managed programming languages use garbage collection (GC) to provide memory safety to applications. Programmers allocate memory and the GC automatically reclaims unused memory. GC strategies are either 'direct', using reference counting to identify and reclaim unreferenced objects, or 'indirect', tracing from known roots to idenitfy reachable objects, and then reclaiming untraced objects. All GC algorithms are graph traversal problems, fundamentally memory-bound, and amenable to parallelization. A GC algorithm may be *concurrent* with respect to the application, and may traverse the graph in *parallel*. GC is therefore prima facie an excellent candidate for heterogeneous hardware.

Interpretation. Most managed languages support dynamic loading and do not perform ahead-of-time compilation. As a consequence, the language runtime must immediately execute code as it is loaded. Interpretation and its variations, including threaded interpretation and template compilation directly to machine code, are normally the default mode of executing code. The interpreter (or template compiler) is highly responsive but offers poor code quality. Advanced virtual machines will identify frequently executed code and dynamically optimize it using an aggressive optimizing compiler. Thus in an advanced runtime, at steady state, performance-critical code is typically optimized and the remaining code executes via interpretation or template compilation. As Figure 1(a) shows, about 20% of all application code executes unoptimized, amounting to about 15% of all cycles. We show that the interpreter is atypical in its microarchitectural requirements, and is significantly better suited to a simple core than optimized application code. Interpretation is thus a prime candidate for execution on a simple core in a power-constrained heterogeneous hardware context. Although interpretation is not inherently parallel, it reflects the parallelism in the application it executes.

Just In Time Compilation. All high performance VM implementations use a JIT optimizing compiler to produce optimized code for frequently executed methods and/or traces. Because a method or trace will have already executed (by an interpreter, for example) at the time the optimizing JIT compiles it,

the runtime has the opportunity to dynamically profile the code, enhancing opportunities for aggressive optimization. A JIT may execute in parallel and asynchronously with the application, according to priorities maintained in a compilation queue. JIT compilation is therefore a natural candidate for exploiting a small core on a big/small multicore.

Compilation strategies may be incrementally more aggressive according to the heat of the target code. Thus, a typical JIT will have several optimization levels. The first level may apply register allocation and common sub-expression elimination and the second level applies optimizations that require more analysis, e.g., loop invariant code motion and loop pipelining. The compiler also performs *feedback directed optimizations*. For example, it chooses which methods to inline at polymorphic call sites based on the most frequently executed target thus far. Similarly, it may perform type specialization based on the most common type, layout code based on branch profiles, and propagate runtime constant values. These common-case optimizations either include code that handles the less frequent cases or that pops out to the interpreter or recompiles when the assumptions fail.

Each system uses a cost model that determines the expected cost to compile the code and the expected benefit. For example, the Arnold et al. cost model used in Jikes RVM uses the past to predict the future [17]. Offline, the VM first computes *compiler DNA*: (1) the average compilation cost as a function of code features such as loops, lines of code, and branches, and (2) average improvements to code execution time due to the optimizations based on measurements on a large variety of code. The compiler assumes that if a method has exectues for 10% of the time thus far, it will execute for 10% of the time in the future. At runtime, the JIT recompiles a method at a higher level of optimization if the predicted cost to recompile it at that level and the reduction in the method's execution time will reduce total time.

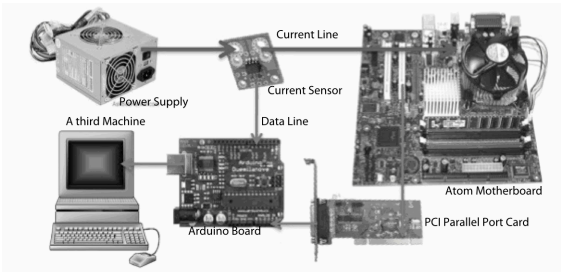
Other VM Services. Other VM services may present good opportunities for heterogeneous multicore architectures, but are beyond the scope of this paper. These include zero-initialization, finalization, and profiling. Managed languages prescribe that memory be initialized by the runtime before being made available to the application. Yang et al. recently showed that 2.7-4.5% of cycles in a Java runtime are dedicated to zeroing memory before it is consumed by the application [18]. They further showed that total execution time can be reduced by 3% on average over 19 benchmarks, by making the zeroing concurrent and utilizing non-temporal store instructions. Zero-initialization may therefore be very well suited to heterogeneous hardware, but we have not evaluated it here. In a managed language, a finalizer is code that, like a destructor, if defined, is executed prior to reclamation of the object. Typically finalization is performed by a distinct finalization thread that works through a queue provided by the garbage collector of unreachable objects that have declared finalizers. Finalization is therefore naturally asynchronous and parallel, and well suited to heterogeneous hardware. Although finalization may in theory be computationally demanding, in practice we found that our benchmarks spent little time performing finalization, so we have not pursed it here. Feedback-directed-optimization (FDO), which is key to the performance of state of the art managed runtimes, depends on profiling of the running application. Such profiling is typically implemented as a producer-consumer relationship, with the instrumented application as the producer and one or more profiling threads as the consumers [19]. The profiler is parallel and exhibits an atypical, memory bound, execution profile, making it a likely candidate for heterogeneous multicore architecture. However, we have not explored profiling in this work.

III. Experimental methodology

1) Hardware: Figure 2(a) lists characteristics of our four experimental machines. Hardware parallelism is indicated in the CMP & SMT row as nCmT: the machine has n cores (CMP) and m simultaneous hardware threads (SMT) on each core. The Atom and Sandy Bridge families are at the two ends of Intel's product line. Atom has an in-order pipeline, small caches, a low clock frequency, and is low power. Sandy Bridge is Intel's newest generation high performance architecture. It has an out-of-order pipeline, sophisticated branch prediction, prefetching, Turbo Boost power management, other optimizations, and

	i7 (32)	i3 (32)	AtomD (45)	Phenom II (45)
Processor	Core i7-2600	Core i3-2120	AtomD510	X6 1055T
Architecture	Sandy Bridge	Sandy Bridge	Bonnell	Thuban
Technology	32nm	32nm	45nm	45nm
CMP & SMT	4C2T	2C2T	2C2T	6C1T
LLC	8MB	3MB	1MB	6MB
Frequency	3.4GHz	3.3GHz	1.66GHz	2.8GHz
Transistor No	995M	504M	176M	904M
TDP	95W	65W	13W	125W
DRAM Model	DDR3-1333	DDR3-1333	DDR2-800	DDR3-1333

(a) Experimental Processors



(b) Hall effect sensor on Atom

Fig. 2. Hardware and Power Measurement Methodology

large caches. We use two Sandy Bridge machines to explore hardware variability, such as cache size, within a family. We choose Sandy Bridge 06_2AH processors because they provide an on-chip RAPL energy performance counter. We also use the AMD Phenom II since it exposes independent clocking of cores to software, whereas the Intel hardware does not.

2) Power and Energy Measurement: This paper uses the Sandy Bridge's on-chip RAPL (Runtime Average Power Limit) energy meters [20], and an improved Hall effect sensor methodology, first developed by Esmaeilzadeh et al. [21] with BIOS configuration. RAPL has three components: power measurement logic, a power limiting algorithm, and memory power limiting control. The power measurement logic uses activity counters and predefined weights to record accumulated energy in MSRs (Machine State Registers). The values in the registers are updated every 1msec, and overflow about every 60 seconds [22]. Reading the MSR, we obtain package, core and uncore energy. We calculate power as energy / runtime. Key limitations of RAPL are (1) only the Sandy Bridge has it, and (2) it can not measure short events, i.e., less than 1msec. Unfortunately, the GC and JIT VM services often occur in phases of less than 1msec.

We extend the Hall effect sensor methodology of prior work to sample more often and use a PCI card to measure finer grain events (i.e., it measures events as short as 1ms) while limiting additional power consumption. Figure 2(b) shows the Pololu's ACS714 Hall effect linear current sensor positioned on the mother board between the power supply and chip. We read the output using an Arduino board with an AVR microcontroller and a sample rate of 5KHz. We connect a PCI card to the digital input of a Arduino board to accurately mark the start and end of VM services. The latency is less then 200μsec. One limitation of this method is that it includes the voltage regulator's power consumption. We compared the Hall effect sensor to RAPL measurements. As expected, power is higher for the Hall effect sensor: 4.8% average, ranging form 3% to 7%. We correct for the voltage regulator by subtracting 5% from Hall effect sensor measurements.

We use relative performance per energy (PPE) and energy to calculate efficiency. The Hall effect sensor measures power, so we calculate energy as the product of power and running time. RAPL presents measures of energy directly. We calculate performance as 1/time for a given workload. We compute relative PPE values based on the experiment. For example, when we compute PPE of hardware feature X compared to feature Y running the same workloads, we calculate:

$$\frac{PPE_X}{PPE_Y} = \frac{\frac{workload_X/run\ time_X}{energy_X}}{\frac{workload_Y/run\ time_Y}{energy_Y}} = \frac{run\ time_Y \cdot energy_Y}{run\ time_X \cdot energy_X}$$

3) Benchmarks: We use 10 benchmarks: bloat, eclipse, and fop (Dacapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (Dacapo-9.12); and pjbb2005 [23]. Fop, luindex and bloat are single threaded. The others are multithreaded. These benchmarks are non-trivial real-world open source Java programs [23].

4) Garbage Collection: We use six garbage collectors in our evaluation to demonstrate the extent to which our findings are sensitive to particular algorithms, and to accommodation methodological challenges. We use three 'stop-the-world' (STW) full heap collectors, the default Immix generational

collector [24] and a concurrent collector configured to run both: a) concurrently, and b) in a special stop-the-world mode. Stop-the-world refers to collection policies that require all application threads to be suspended while collection is performed. We envisage collection performed concurrently with the application on future big/small multicores. We include STW collectors in our analysis because: a) the pause-times of the generational collector are often less than our measurement granularity of 200μsec, b) these heap structures reflect existing collectors, (c) existing concurrent GCs have not been tailored to this setting so are unrealistically inefficient, and d) we have no way of measuring the power and/or energy of a particular thread, we cannot directly measure power and/or energy of a concurrent collector.

All of our evaluations are performed within Jikes RVM's memory management framework, MMTk [25], [24]. We report time for the default Immix generational collector in Figure 1(a) because it is the best performing collector and thus yields the lowest time. We use the three STW collectors for architectural sensitivity studies since they reflect the three most widely-used base algorithms. Each makes distinct time-space tradeoffs. Our concurrent collector uses a classic snapshot-at-the-beginning algorithm [26]. Because GC is a time-space tradeoff, the available heap space determines the amount of work the GC does, so it must be controlled. Section IV-A uses a heap 1.5 $\times$ the minimum in which the collectors executes and is typical. For the concurrent collector, the time-space tradeoff is significantly more complex because of the concurrency of the collector's work, so we explicitly controlled the GC workload by forcing regular concurrent collections every 8MB of allocation for avrora, fop, and luindex, which have low total allocation, and 128MB for the remaining benchmarks. Unlike the STW collectors, we cannot isolate the concurrent collector, so we establish its performance and energy consumption by subtraction. We take the total execution time when using the concurrent collector and subtract from that the non-GC component of the STW concurrent collector.

5) Interpreter: Evaluating the interpreter is challenging because interpretation is interwoven with optimized code execution. We evaluate the microarchitectural sensitivity of the interpreter by executing the benchmarks on the Oracle HotSpot JDK 1.6.0 and Jikes RVM. HotSpot interprets bytecodes and Jikes RVM template compiles them. Both adaptively (re)compile hot methods to machine code. The HotSpot experiment uses the default GC (parallel copying nursery and serial Mark-Sweep-Compact old generation) and heap size configured with `-server` flag. To understand the interpreter's response to hardware variations, we turn the JIT off so all application code was executed via the interpreter. To estimate the fraction of cycles spent in the interpreter, we sample execution in Jikes RVM (via its existing timer-based sampling mechanism) and then count the fraction of samples in non-optimized versus optimized code.

6) JIT Compiler: Because the unit of work for the JIT when executing normally is too fine grained for RAPL energy or the Hall sensor power measurements, we perform and measure all JIT work at once from a *replay* profile. *Replay compilation* removes the nondeterminism of the adaptive optimization system. It gathers a compilation profile on a previous execution that dictates what the adaptive compiler chose to do. We execute the benchmark again. We then apply and measure the JIT optimizing compilation that exactly applies the profile all at once. We then turn off the JIT and measure the application on the second iteration. The application thus contains the same mix of optimized and unoptimized code as it would have eventually had in vivo execution, but now we can measure both the compiler and application independently and thus the experiments are repeatable and measurable with small variation. To decrease or eliminate GC when measuring the JIT, we use Jikes RVM's default generational Immix GC, since it performs the best, and set the heap size to be 4 times minimum generational Immix GC size. In summary, we follow Blackburn et al.'s best practices for Java performance analysis [23] and adopt it to deal with the limitations of the power and energy measurement methodologies.

7) Small Core Evaluation: To understand the amenability of the various services to a small core microarchitecture, we use the i3 and Atom processors shown in Figure 2(a). The processors differ in two ways that are inconsistent with a single-die setting: a) they have different process technologies (32nm

v 45nm), and b) they have different memory speeds (1.33GHz v 800MHz). Our comparisons adjust for both by down-clocking the i3's memory speed to match the Atom, and by approximating the effect of the technology shrink. Esmaeilzadeh et al. find that a die shrink from 45nm to 32nm reduces processor energy and power by 45% on two Intel architectures [21]. We use the same factor, but do not adjust for core clock speed, on the grounds that a simple low power core may well run at a lower frequency.

To evaluate the overall power and PPE effects of deploying the JIT on a slower core in Section IV-C, we use the AMD Phenom II described in Figure 2(a). This processor supports independent clocking of cores and a wide range of clock frequencies, down to 800MHz, less than 1/3 the default clock speed of 2.8GHz. Because the Phenom II does not support RAPL, we use the Hall effect sensor methodology to establish its power and energy.

8) Microarchitectural Characterization: We evaluate the effect of *frequency scaling* on the i7, varying the clock from 1.6GHz to 3.4GHz, and plotting performance, energy, and power as a function of clock frequency, normalized to 1.6GHz. To understand sensitivity to *memory bandwidth*, we use the i3 and compare energy, power, performance and PPE with 800MHz, single channel memory relative to the default 1.33GHz, dual channel memory. We explore the effect of SMT and CMP *hardware parallelism* by using the i7 configured as 1C2T and 2C1T respectively, and then compare against the i7 configured as 1C1T. We evaluate the effect of different *last level cache* sizes by comparing the i7 and i3, each configured to use two cores at 3.3GHz, but with 8MB and 3MB of LLC respectively. To understand the effect of *gross microarchitectural change*, we compare the i3 and Atom running at the same clock speed, and make adjustments for variation in process technology, reducing the energy and power of the Atom by 45% to simulate fabrication at 32nm [21].

IV. Experimental Analysis

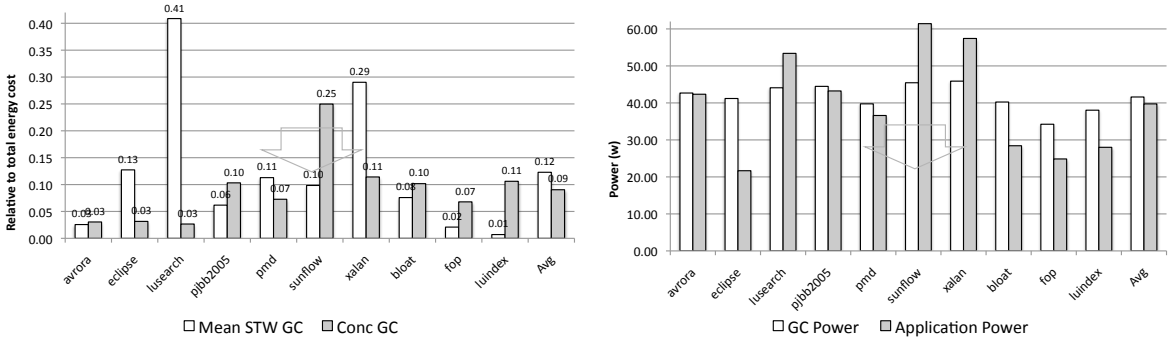
We now present a quantitative analysis of garbage collection, interpretation and JIT compilation with respect to their suitability to heterogeneous multicore architectures. Power is presented in absolute terms (W), averaged over time for the duration of the execution of the given service. Because interpretation is more finely interleaved than our power meter can resolve, we do not present power data for the interpreter. In each of the other graphs the data is normalized, with large transparent arrows indicating the direction of improvement: energy (lower), power (lower), performance (higher), PPE (higher).

A. Garbage Collection

We start with garbage collection. The software characteristics of garbage collection make it an excellent candidate for a big/small multicore. Collection is *parallel*, *asynchronous* with respect to the application, is only on the application's *critical path* when memory is scarce, and is *hardware sensitive*.

We measure concurrent garbage collection, which can execute in parallel with the application; three classic stop-the-world (STW) collectors; and a stop-the-world variant of the concurrent collector. We use these later configurations to demonstrate the robustness of our findings across algorithms and due to methodological necessity. See Section III-4 for details. The following graphs present the mean across the four STW collectors ('Mean STW GC'), and where methodologically feasible, the concurrent collector ('Conc GC'). We present results for all non-GC computation ('Application'), measured as the non-GC portions of total execution when using STW collectors. Because garbage collection is parallel, we perform our evaluation using all 8 hardware contexts (4C2T) available, by default.

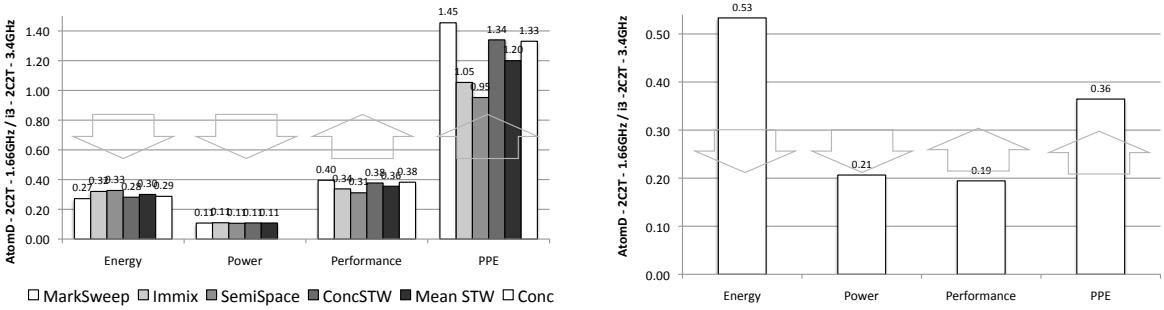
1) Power and Energy Footprint: Figure 3(a) shows the fraction of total energy devoted to garbage collection for both STW and concurrent GC. The results indicate considerable variation between STW and concurrent, but both yield similar averages of 12% and 9%, respectively. The variation across benchmarks reflects how the different collection heuristics pan out on various workloads. Figure 3(b) shows the power consumption of the STW GCs and application running on the i7. There is less variation in power across benchmarks and on average power consumption of the GC and application are similar on this large out-of-order processor. The data in Figure 3(a) confirms our hypothesis that garbage collection is an



(a) Relative GC Energy.

(b) GC and application power cost.

Fig. 3. GC power and energy cost on i7 4C2T at 3.4GHz



(a) Garbage Collection

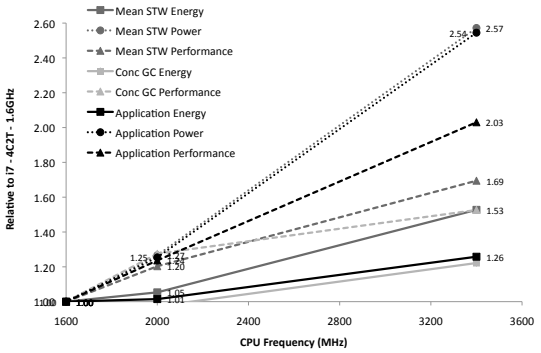
(b) Total Application

Fig. 4. Power and energy savings on simple core for GC (a) and the application (b).

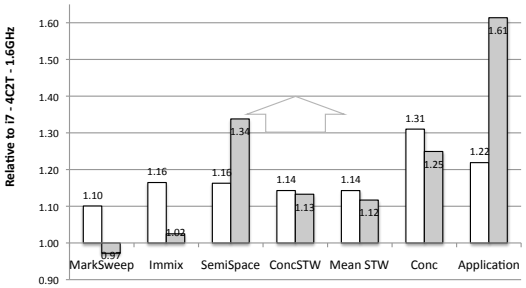
important target for reducing energy, while Figure 3(b) shows that GC is a significant consumer of power when run on a large core.

2) Amenability to Simple Core: Figure 4 illustrates the amenability of GC to a simple core, by showing in Figures 4(a) and 4(b) the effect on energy, power, performance and PPE of moving the GC and application (4(a) and 4(b) respectively) from a big core to a small core. We use an i3 and AtomD with the same degree of hardware parallelism (2C2T), and run with the same memory bandwidth. We do not adjust for clock frequency in this experiment on the grounds that the small core may well be run at a lower clock, so they run at 3.4GHz and 1.66GHz respectively. We explore the effect of clock scaling separately, below. The most important difference in the two machines is their process technology. Esmaeilzadeh et al. show that shrinking technology from 45nm to 32nm for the same microarchitecture and controlling for frequency decreases power by 45% or more, while the providing the same performance [21]. To estimate the effect of shrinking the process, we project the AtomD power consumption data to 32nm by multiplying with a projection factor of 55%.

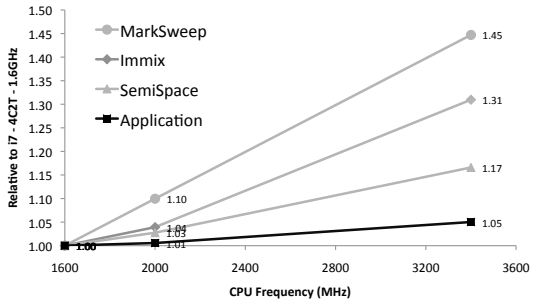
Figure 4(a) shows each of the four STW collectors, their mean, and the concurrent collector. The data is quite consistent across collectors and shows that the simple core (in order and low frequency) can save about 70% energy and 90% power compared to the i3. The bottom line is a significant *improvement* to



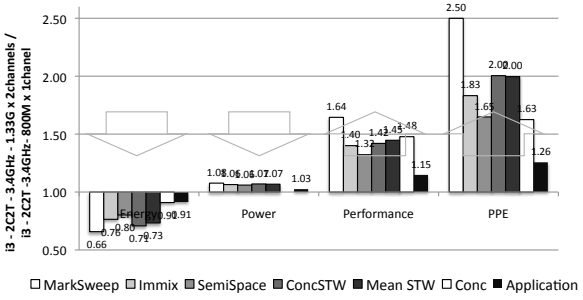
(a) Effect of clock frequency on GC and application.



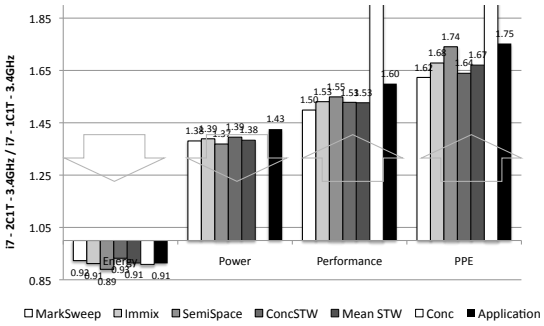
(b) PPE for GC and application at 2.0GHz (light) and 3.4GHz (dark) compared to 1.6GHz.



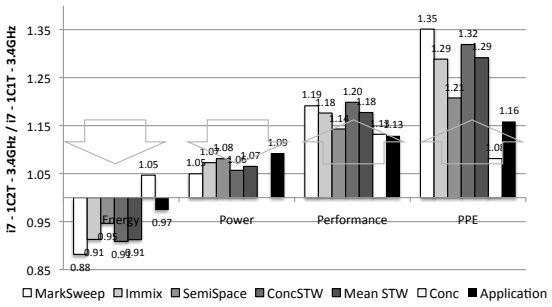
(c) Memory stalls result in more cycles executed.



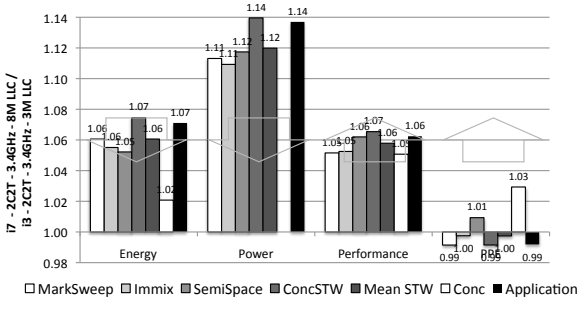
(d) Memory bandwidth effect on GC and application.



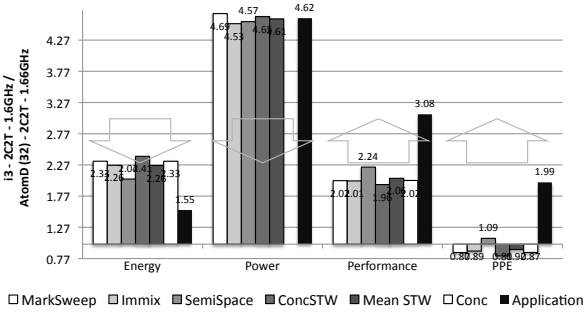
(e) CMP effect on GC and application.



(f) SMT effect on GC and application.



(g) LL cache size effect on GC and application.



(h) Gross architecture effect on GC and application.

Fig. 5. Microarchitectural characterization of GC.

PPE for three out of five of the collectors, including the concurrent collector (33%). By contrast, Figure 4(b) shows a significant 64% *degradation* to PPE when running the non-GC computation on the simple core. This contrasting outcome is due both to a bigger power reduction (90% compared to 80%), and a smaller performance reduction (60% compared to 80%) for the GC. This data makes it emphatically clear that the simple memory-bound graph traversal at the heart GC is much better suited to small, low power in-order processors.

3) Characterization of GC: We now explore the sensitivity of the garbage collectors to a variety of hardware features.

Frequency scaling. Figures 5(a)-5(c) explore the effect of clock frequency on the garbage collectors and the application. We evaluate the systems on the i7 at 1.6, 2.0, and 3.4GHz. In Figure 5(a), we see that STW GC power and application power (solid circles) respond almost identically, with about 250% increase in power as the clock increases from 1.6 to 3.4GHz (note that we cannot directly measure power for the concurrent GC). However, performance does not increase as rapidly for the GCs (69% and 53%) as it does for the application (103%). Total energy consumption rises more dramatically for the STW GCs (53%) than for the concurrent GC and the application (26%).

In Figure 5(b) we see how these responses translate into PPE, with the collectors experiencing much lower PPE improvements than the application (61%), and the mark sweep collector even experiences a small PPE *degradation* at 3.4GHz compared to the PPE at 1.6GHz. Among the STW GCs, relative GC performance differs by over 22% on average. SemiSpace is worst, then MarkSweep, and Immix is best [24]. Increasing frequency is inefficient because GC has poor locality and accesses memory at a higher rate than the average application. Hardware performance counters reveal that SemiSpace's last level (LL) cache misses per instruction are 3+ times greater than for the applications Immix GC is 5+ times; and MarkSweep is 10+ times. Figure 5(c) shows that the stalls induced by these memory accesses are increasingly evident as the clock frequency rises. The relative number of cycles increases from 17% to 45% for GC, but only by 5% for the application.

Memory bandwidth. Figure 5(d) shows that increasing memory bandwidth provides substantial performance and consequently energy improvements for memory-bound GC. The PPE increase for GC ranges from 66% (SemiSpace) to 150% (MarkSweep) with a STW average of 200%. Interestingly these numbers clearly show that the concurrent GC is *far* less sensitive to memory bandwidth than the STW GCs. This is unsurprising because the concurrent GC amortizes its graph traversal over much longer periods of time, while the STW GCs must perform their graph traversals in sharp bursts, maximizing their exposure to memory bandwidth constraints.

Hardware parallelism. We study the effects of hardware parallelism on the GC and the application using the i7 with 2C1T and 1C1T at 3.4GHz for CMP, 1C2T and 1C1T for SMT. All the GC algorithms are parallel and utilize all available hardware parallelism. For the application results, we use the six benchmarks that are multithreaded. Figure 5(e) shows that increasing the number of cores improves both GC and multithreaded applications' PPE similarly, with the notable exception of the concurrent collector, which benefits disproportionately from the extra core. For STW GC, doubling the number of cores decreases energy by 9%, increases performance by 53%, and thus improves PPE by 67%. Figure 5(f) shows that SMT does not effect PPE as much as CMP, but effectively decreases energy. SMT requires very little additional power as compared to CMP, which is important when the power budget is limited. Among the STW collectors, MarkSweep benefits most from SMT while SemiSpace benefits most from CMP. This is unsurprising since MarkSweep is the memory bound, so it can benefit from the latency hiding offered by SMT. On the other hand SemiSpace is inefficient [25], so makes good use of the additional core to reduce its computational load.

Last-level cache size. One way to use the abundant transistors on chip is to increase cache size. This experiment evaluates the approximate effect of increased cache size on the GC and application using the i7 and i3. We configure them with the same hardware parallelism (2C2T) and clock frequency (3.4GHz).

After controlling for hardware parallelism and clock speed, the most conspicuous difference between the systems is their last level cache: the i7's LL cache is 8MB and i3's is 3MB. For this experiment we choose four benchmarks with large minimum heap sizes of more than 100MB (bloat, eclipse, pmd and pjbb2005) to ensure that the live object set is significantly larger than LL cache size. For these benchmarks, the increase in LL cache size from 3MB to 8MB is approximately 3% of the average maximum volume of live objects. Noting the y-axis scale, Figure 5(g) shows that none of GCs or the application are particularly sensitive to LL cache size.

Gross microarchitecture. Figure 5(h) compares the impact of gross microarchitecture using the AtomD and i3, controlling for frequency (1.66G for AtomD and 1.6G for i3), hardware parallelism (2C2T), and memory bandwidth (800 MHz, one channel). We adjust for process technology as we do above by scaling by 55%.

The results make it clear that the high performance architectural features of the i3 are nowhere near as effective for GC as they are for the application. Compared to AtomD, the i3 benefits applications significantly (PPE doubles), but actually degrades PPE for the GCs. It is clear from the performance data that the large out of order core is far less effective at delivering performance to any of the GC algorithms as it is at delivering performance to the application.

B. Interpretation

The software characteristics of the interpreter include that it is on the application's critical path, and because it is tightly coupled with the application it exhibits no independent parallelism and cannot execute asynchronously. However, parallelism within the application and between the application and the VM offer the interpreter opportunities to exploit parallelism. For example, in a single threaded application the interpreter could continue executing a loop on a small core while the compiler uses a large core to optimize the same loop. Or, in a multithreaded application, threads running interpreted code could utilize small cores while threads running optimized code execute on large cores. This section shows that the interpreter is sensitive to clock speed, but not to other high performance hardware features.

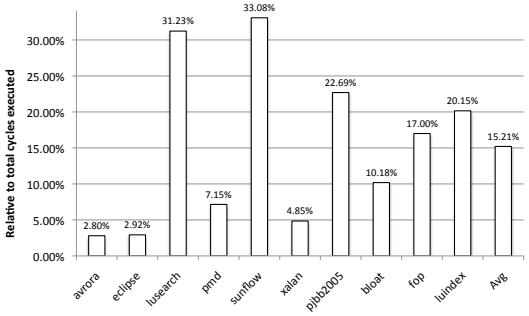


Fig. 6. Fraction of cycles executing unoptimized code.

1) Power and Energy Footprint: The fine-grained interleaving of the interpreter and the application code also makes it impossible to directly measure the power and energy of the interpreter. For this reason and because Oracle JDK does not expose much of its measurement infrastructure, we are unable to directly measure the fraction of cycles consumed by the Oracle JDK's interpreter. We instead use Jikes RVM's built-in profiling mechanisms to measure the fraction of cycles spent executing unoptimized code generated by its template compiler, that VM's equivalent to an interpreter. Figure 6 shows this percentage, providing an estimate of the fraction of cycles spent in the interpreter and a first order indication of the interpreter's impact on energy. On average 15%, and as much as 33% of cycles are spent executing unoptimized code. This same data appears in the 'Interpreter' component of Figure 1(a).

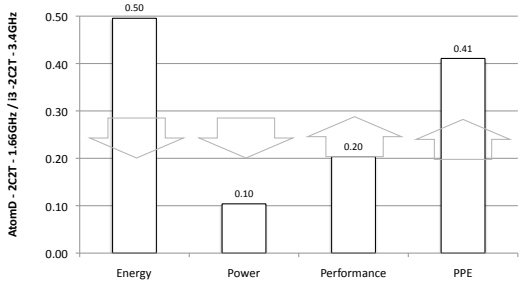


Fig. 7. Interpreter power & energy savings on simple core.

2) Amenability to Simple Core: Figure 7 shows the power, performance, energy, and PPE effect on the interpreter of executing on the AtomD and i3. Figure 4(b) shows the effect on the application. A

simple core offers a 90% power saving to the interpreter! This result is significantly better than the 79% saving on the entire application. The interpreter's performance is very sensitive to clock speed, so the performance of the interpreter is impacted more heavily than the GC, leading to a less impressive energy improvement of 50% and a degradation in PPE of 60%, slightly better than the total application (64%). The simple core is clearly a power and energy efficient execution context for the interpreter in contexts when the performance reduction is tolerable. Since performance-critical code is typically optimized and therefore not interpreted, it is plausible that interpreted code is less likely to be critical and therefore more resilient to execution on a slower core.

3) Characterization of Interpreter: We now characterize the interpreter with respect to major hardware parameters. To do this we eliminate compiled code and minimize the garbage collector by executing in the HotSpot VM with compilation disabled and use a very large heap. In the graphs that follow we also plot the effect upon the 'Application', which we represent using the steady state performance of Jikes RVM with GC costs subtracted.

Frequency scaling. Figure 8(a) shows that the interpreter scales even better with clock frequency than normal applications. We measured the LL cache access rate for the interpreter and applications; it is 18% on average, compared to 29% for applications. Increasing the clock rate thus benefits the interpreter more than the application because the additional memory accesses become relatively more expensive as the clock speed increases.

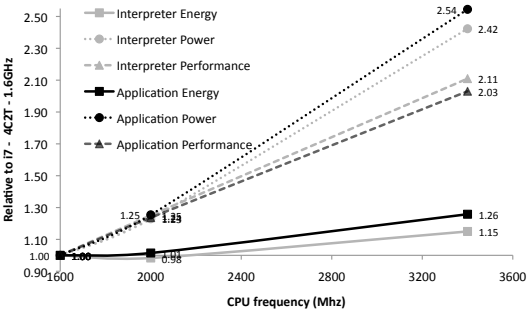
Hardware parallelism. Remarkably, both SMT and CMP hardware parallelism improve PPE when the application executes exclusively with the interpreter. Figures 8(c) and 8(d) show that the use of CMP decreases energy by 25% and SMT decreases energy by 20%. The PPE increases for CMP and SMT are 146% and 77%; both are much higher than the improvements for the application or GC. These improvements come from two sources: parallelism in the application (7 of the 10 applications are multithreaded) and parallelism in the VM (as shown previously [21]). The interpreter exploits SMT parallelism very well (43% performance improvement compared to 13%), with a modest increase in power (16% compared to 9%). On the other hand, CMP also improves interpreter performance very well, but relatively less when compared to the application (84% compared to 60%) but does so with less power overhead (35% compared to 43%). Five of the nine benchmarks saw the most dramatic performance improvements on the SMT and CMP respectively: xalan (54%, 113%), avrora (55%, 74%), pmd (45%, 113%), lusearch (53%, 111%), and pjbb (36% and 67%). All of these benchmarks are multithreaded, while three out of the remaining benchmarks are single threaded. Eclipse is the only multithreaded benchmark that did not improve due to hardware parallelism when fully interpreted.

Memory bandwidth and Last-level cache size. Figures 8(b) and 8(e) further show that the interpreter has good locality — it is insensitive to both cache size and memory bandwidth. Because the larger cache size offers no performance advantage but consumes power, it will degrade the PPE of the interpreter.

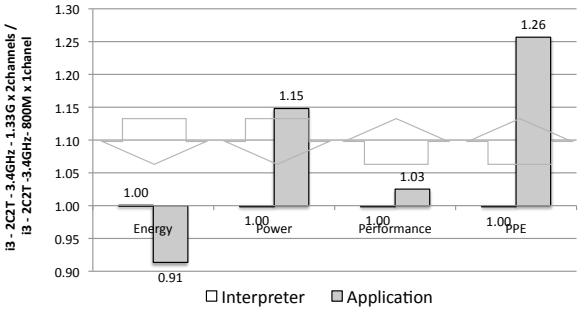
In summary, the interpreter benefits from hardware parallelism and a high clock rate, but none of the other high-performance core features. Executing the interpreter on a small core degrades performance some, but provides substantial energy improvements.

C. Just In Time Compilation

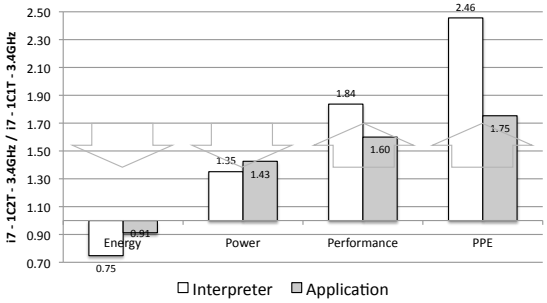
The software characteristics of the Just in Time (JIT) compiler include that it exhibits *parallelism* and may execute *asynchronously*. It can contribute to the application's critical path, if it does not deliver optimized code in a timely manner, or if it competes for on the same processor. JIT compilation is typically driven by a work queue of code to optimize. The actually JIT compilation can operate asynchronously with the application execution and if multiple JIT threads operate on the work queue, it provides software parallelism. Modern VMs use both single and multi-threaded JITs. The Jikes RVM JIT is single threaded. As explained in Section III-6 to measure JIT power, we must perform JIT compilation en mass rather than incrementally. Since the Jikes RVM JIT is not parallel, for all but Figure 10(b), we evaluate the JIT on a single hardware context (1C1T).



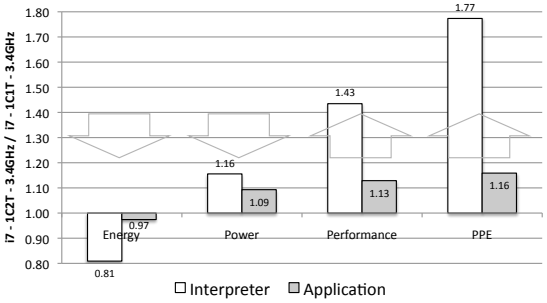
(a) Frequency effect on interpreter and application.



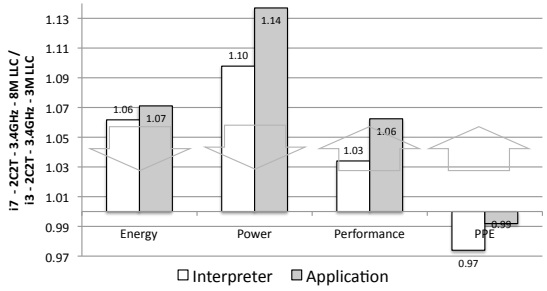
(b) Memory bandwidth effect on interpreter and application.



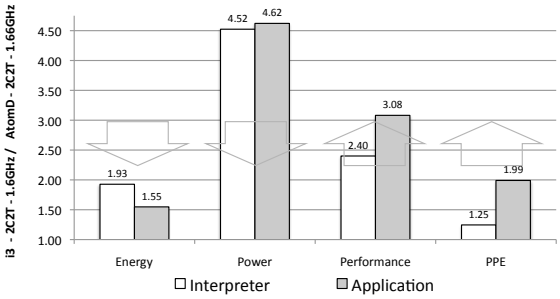
(c) CMP effect on interpreter and application.



(d) SMT effect on interpreter and application.



(e) LL cache size effect on interpreter and application.

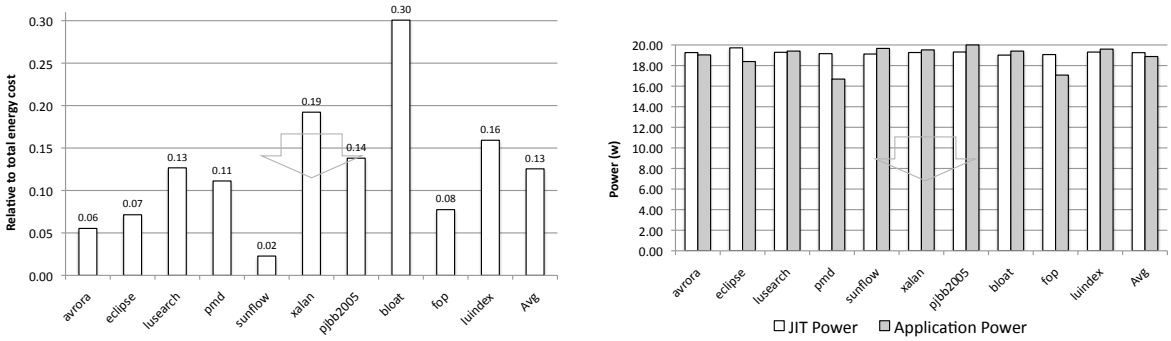


(f) Gross architecture effect on interpreter and application.

Fig. 8. Microarchitectural characterization of the interpreter.

1) JIT Power and Energy Footprint: Figure 9 shows the relative power and energy consumption of the JIT. The JIT is a significant consumer of energy, using 13% of all energy, on average, and as much as 30%, while Figure 9(b) shows that the JIT draws very similar power to the application.

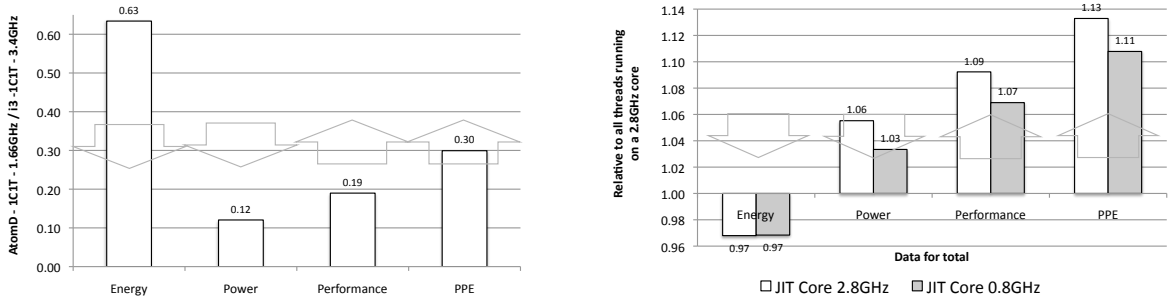
2) Amenability of JIT to Simple Core: Figure 10(a) shows that a small core reduces the JIT's power consumption by nearly 90% and its energy consumption by nearly 37%. Unfortunately, performance is degraded by just over 80%, resulting in a PPE degradation of 70%. However, because the JIT is not on the critical path and can work asynchronously with the application, performance need not be a first order concern. We evaluate this hypothesis in Figure 10(b) on an orthodox AMD Phenom II multicore, by binding the JIT to a separate core and substantially under clocking the JIT core, while all other threads are bound to the full speed core. These results show that offloading the JIT to a separate core improves performance by 7% to 9% for the slow and fast cores respectively. In this configuration, the JIT is taken off the application's critical path. Running the JIT on a separate core also slightly increases power, but



(a) JIT energy relative to application.

(b) JIT and application power.

Fig. 9. JIT power and energy cost on the i7 1C1T @ 3.4GHz.



(a) JIT energy and power savings on a simple core.

(b) Overall effect when the JIT is bound to a separate core running @ 2.8GHz or 0.8GHz on an AMD Phenom II.

Fig. 10. Amenability of JIT to executing on a separate, slower core.

it reduces energy marginally and improves PPE by 11-13%.

3) Characterization of JIT: Frequency Scaling, Memory Bandwidth, and *Gross Microarchitecture.* Figures 11(a), 11(b), and 11(d) show that the JIT and the application are remarkably similar in their power, performance, and energy responses to these architectural changes. Because Jikes RVM's JIT is single-threaded, we could not evaluate the JIT's response to CMP or SMT hardware parallelism. The characteristics of compilers have been studied extensively in the literature and are included in many benchmark suites including the ones we use here, so we did not find this similarity surprising. *Last-level cache size.* Interestingly, the JIT benefits from a larger last-level cache, with a 10% improvement in PPE, while our applications are slightly degraded, on average. The JIT is thus the only context in which we see a PPE benefit from an increase in last level cache size.

4) Further Opportunities for the JIT: Executing the JIT on small cores offers a new opportunity to improve code quality. Minimizing JIT cost means that in theory, the JIT can optimize more code, more aggressively, improving application code quality. Figure 12 evaluates this hypothesis. It compares the total performance (GC, application, and interpreter, but not JIT) using the standard optimization decisions that Jikes RVM produces on the first iteration using its cost benefit computation (see Section II), to a more aggressive cost model model. We configure the compiler cost model by reducing the cost of compilation by a factor of 10. Figure 12 shows that making the JIT more aggressive improves total

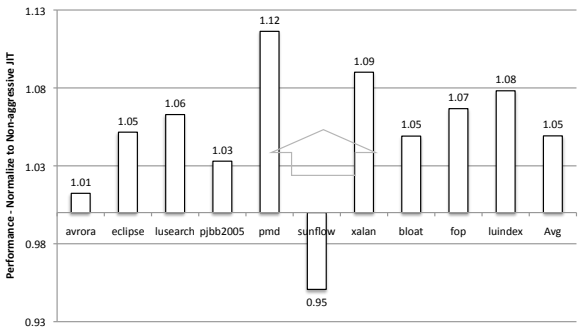
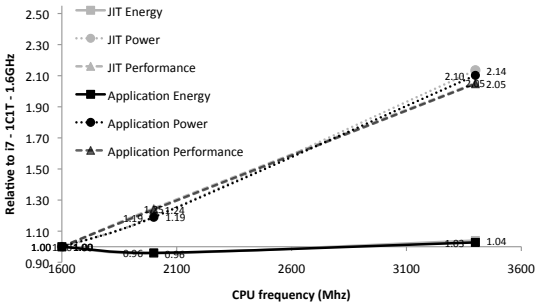
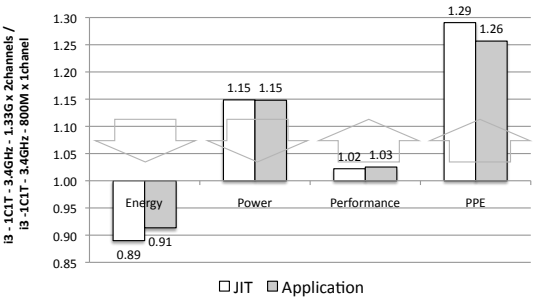


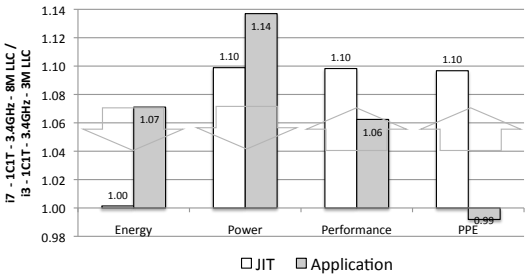
Fig. 12. Total performance improvement due to more aggressive JIT.



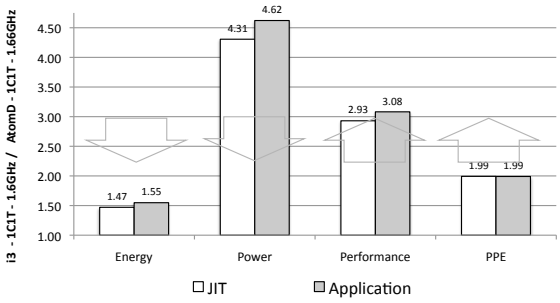
(a) Frequency scaling: effect on JIT and application.



(b) Memory bandwidth: effect on JIT and application.



(c) Last-level cache size: effect on JIT and application.



(d) Gross architecture: effect on JIT and application.

Fig. 11. Microarchitectural characterization of the JIT.

performance by around 5%. This result suggests that software/hardware co-design has the potential to improve power, performance, and PPE further.

Discussion. To realize all these gains requires more research and infrastructure. At least, we will need (1) big/small core hardware with new hardware features, e.g., voltage scaling on a thread basis to for example accelerate the interpreter; (2) on-chip power meters, (3) OS scheduling support; (3) concurrent GC and parallel JIT tuned for the hardware; and (4) total system algorithms to coordinate application and VM threads.

V. Related work

Except for early LISP machines [27], most architecture research for managed languages has focused on GC energy and hardware, not on JIT or interpretation. Velaso et al. study the energy consumption of state-of-the-art GCs for designing embedded systems [28]. They use Jikes RVM, Dynamic SimpleScalar (DSS) [29], and combine DSS with a CACTI energy/delay/area model to calculate energy. Their energy simulation results follow the performance measurements from prior work [25]. However, their simulation results show GC consumes a disproportionate amount of energy, but our results refute this finding. Chen et al. study mark-sweep GC using an energy simulator and the Shade SPARC simulator [30]. They improve leakage energy by using a GC-controlled optimization to shut off memory banks that do not hold live data. Diwan et al. measure the impact of four memory management strategies on C programs executing on the Itsy Pocket Computer [31]. Their results demonstrate that the memory management algorithm changes the program's energy consumption. Our paper focuses more broadly on hardware customization for VM services, rather than improving energy with GC or selecting an energy-efficient GC algorithm.

Meyer et al. explore hardware for GC [32][33][34]. They develop a novel processor architecture with an objected-based RISC core and a GC core using VHDL and generate an FPGA. Their goals are to eliminate GC pauses for real-time embedded systems and improve safety and reliability given stringent safety requirements, such as satellites and aircrafts. In comparison, we focus on general purpose hardware

and software. Azul systems built a custom chip to run Java business applications and severs [35]. They redesign about 50% of CPU and build their own OS and VM. The chips have special GC instructions, such as read and write barriers, but do not have specialized GC cores. Our work focuses more broadly on VM services and explores if tuning general-purpose multicore hardware can improve overall PPE.

VI. Conclusion

Rapid change in hardware design is relentless. However changes afoot today in research and industry are increasingly less predictable and more complex. Furthermore, many application developers have chosen managed languages, insulating them from hardware change by a layer of abstraction. One emerging hardware trend is single ISA heterogeneous multicore. This paper shows that a software/hardware co-design of big/small multicore and system software, and in particular VM services, is a promising approach. Our data suggests that very simple cores tailored to services such as garbage collection, the interpreter and JIT will be very effective. By targetting VM services, our approach has broad reach, does not burden application programmers, nor does it compromise application portability to rapidly evolving hardware.

References

[1] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ASPLOS*, J. C. Hoe and V. S. Adve, Eds. ACM, 2010, pp. 205–218.

[2] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, R. Iyer, Q. Yang, and A. González, Eds. ACM, 2011, pp. 365–376.

[3] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM/IEEE International Symposium on Computer Architecture*, 2007, pp. 13–23.

[4] E. P. Agency, "Report to congress on server and data center energy efficiency," U.S. Environmental Protection Agency, pp. 109–431, 2007.

[5] Wikipedia, "Google platform." 2009. [Online]. Available: http://en.wikipedia.org/wiki/Google_platform

[6] J. G. I. P. Council. (2011) Concept of new metrics for data center energy efficiency. [Online]. Available: http://www.greenit-pc.jp/e/topics/release/100316_e.html

[7] ARM Corporation, "big.LITTLE processing," 2011. [Online]. Available: http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php

[8] T. Y. Morad, U. C. Weiser, A. Kolodnyt, M. Valero, and E. Ayguadé, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *Computer Architecture Letters*, vol. 5, no. 1, pp. 14–17, 2006.

[9] S. Borkar and A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[10] T. Li, P. Brett, R. C. Knauerhase, D. A. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *International Conference on High-Performance Computer Architecture*, 2010, pp. 1–12.

[11] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, Paris, France, 2010, pp. 125–138.

[12] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multicore architectures," *IEEE Micro*, vol. 30, no. 1, pp. 60–70, 2010.

[13] D. Bacon, "Virtualization in the age of heterogeneous machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2011, pp. 1–2.

[14] 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 *MICRO*. ACM/IEEE, 2003, pp. 81–92.

[15] Texas Instruments, "Omap 5 platform," 2011. [Online]. Available: http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12864&contentId=103103

[16] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley, "Microarchitectural characterization of production JVMs and Java workloads," in *IBM CAS Workshop*, Austin, Texas, 2008.

[17] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, "Adaptive optimization in the jalapeño jvm (poster session)," in *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*. ACM, 2000, pp. 125–126.

[18] X. Yang, S. Blackburn, D. Frampton, J. Sartor, and K. McKinley, "Why nothing matters: The impact of zeroing," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. ACM, 2011, pp. 307–324.

[19] J. Ha, M. Arnold, S. Blackburn, and K. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 155–174.

[20] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le, "Rapl: Memory power estimation and capping," in *ISLPED*, V. G. Oklobdzija, B. Pangle, N. Chang, N. R. Shanbhag, and C. H. Kim, Eds. ACM, 2010, pp. 189–194.

[21] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley, "Looking back on the language and hardware revolutions: measured power, performance, and scaling," in *ASPLOS*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 319–332.

[22] I. Coorporation, "Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1," pp. 643–655, 2011.

[23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Oct. 2006, pp. 169–190.

[24] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality," in *Programming Language Design and Implementation*, Tuscon, AZ, Jun. 2008, pp. 22–32.

[25] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *SIGMETRICS*, NY, NY, Jun. 2004, pp. 25–36.

[26] T. Yuasa, "Real-time garbage collection on general-purpose machines," *Journal of Systems and Software*, vol. 11, pp. 181–198, March 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=82232.82237

[27] R. D. Greenblatt, T. F. Knight, J. T. Holloway, and D. A. Moon, "A lisp machine," in *The Papers of the Fifth Workshop on Computer Architecture for Non-Numeric Processing, Pacific Grove, CA, USA, March 11-14, 1980*, vol. 10, no. 4. ACM, 1980, pp. 137–138.

[28] J. Velasco, D. Atienza, K. Olcoz, F. Catthoor, F. Tirado, and J. Mendias, "Energy characterization of garbage collectors for dynamic applications on embedded systems," *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pp. 908–915, 2005.

[29] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger, "Dynamic SimpleScalar: Simulating Java virtual machines," University of Texas at Austin, Department of Computer Sciences, Technical Report TR-03-03, Feb. 2003.

[30] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded java environment," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002, pp. 92–103.

[31] A. Diwan, H. Lee, D. Grunwald, and K. Karkas, "Energy consumption and garbage collection in low-powered computing," University of Colorado, Boulder, Technical Report CU-CS-930-02, 2002.

[32] O. Horvath and M. Meyer, "Fine-grained parallel compacting garbage collection through hardware-supported synchronization," in *ICPP Workshops*, W.-C. Lee and X. Yuan, Eds. IEEE Computer Society, 2010, pp. 118–126.

[33] S. Stanchina and M. Meyer, "Mark-sweep or copying?: a "best of both worlds" algorithm and a hardware-supported real-time implementation," in *ISMM*, G. Morrisett and M. Sagiv, Eds. ACM, 2007, pp. 173–182.

[34] ——, "Exploiting the efficiency of generational algorithms for hardware-supported real-time garbage collection," in *SAC*, Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, Eds. ACM, 2007, pp. 713–718.

[35] C. Click, "Azuls experiences with hardware/software co-design. keynote presentation at the acm sigplan," in *SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.