The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software

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

***Abstract*—On the *hardware* side, asymmetric multicore processors present software with the challenge and opportunity of optimizing in two dimensions: performance and power. Asymmetric multicore processors (AMP) combine general-purpose *big* (fast, high power) cores and *small* (slow, low power) cores to meet power constraints. Realizing their energy efficiency opportunity requires workloads with differentiated performance and power characteristics.**

On the *software* side, managed workloads written in languages such as C#, Java, JavaScript, and PHP are ubiquitous. Managed languages abstract over hardware using Virtual Machine (VM) services (garbage collection, interpretation, and/or just-in-time compilation) that together impose substantial energy and performance costs, ranging from 10% to over 80%. We show that these services manifest a differentiated performance and power workload. To differing degrees, they are parallel, asynchronous, communicate infrequently, and are not on the application's critical path.

We identify a synergy between AMP and VM services that we exploit to attack the 40% average energy overhead due to VM services. Using measurements and very conservative models, we show that adding small cores tailored for VM services should deliver, at least, improvements in performance of 13%, energy of 7%, and performance per energy of 22%. The *yin* of VM services is overhead, but it meets the *yang* of small cores on an AMP. The *yin* of AMP is exposed hardware complexity, but it meets the *yang* of abstraction in managed languages. VM services fulfill the AMP requirement for an asynchronous, non-critical, differentiated, parallel, and ubiquitous workload to deliver energy efficiency. Generalizing this approach beyond system software to applications will require substantially more software and hardware investment, but these results show the potential energy efficiency gains are significant.

I. Introduction

Computer hardware is facing a power crisis. For mobile devices, battery life always falls behind demand. For data centers, the cost of electricity is a top budgetary consideration [1], [2]. Consequently, large companies purchase computers with the best performance per energy (PPE) dollar, rather than the best absolute performance [3], and the Japanese Green IT Council promotes PPE as a world standard metric for data center efficiency [4]. To lower power and improve PPE, architects are turning to customization and heterogeneity [5], [6], [7], [8], [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.

This research was supported by ARC DP0666059, NSF CSR-0917191, the 863 program of China 2012AA010905, and the National Natural Science Foundation of China 61070037, 61103016.

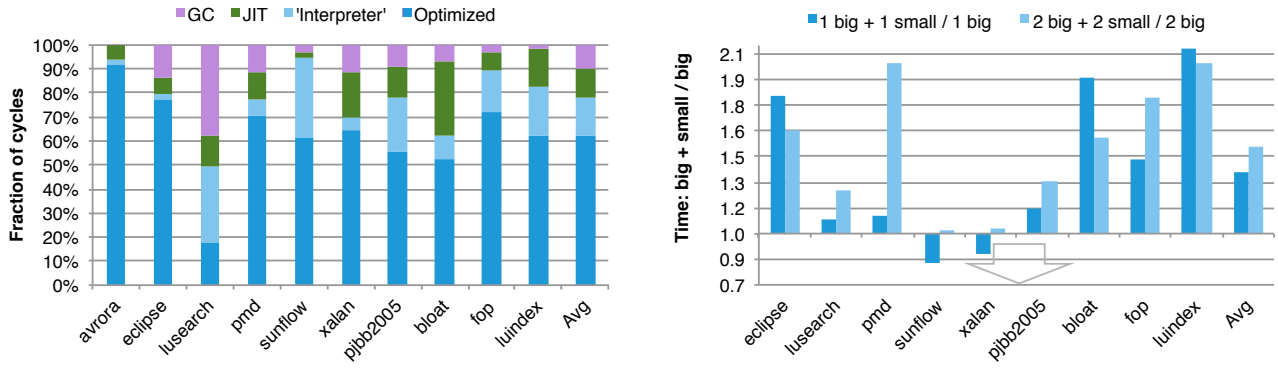
Meanwhile, software is facing challenges of a similar magnitude, with major changes in the way software is deployed, is sold, and interacts with hardware. Developers are increasingly choosing managed languages, sacrificing performance for programmer productivity, time-to-market, reliability, security, and portability. Smart phone and tablet applications are exclusively written in managed languages. Modern web services combine managed languages, such as PHP on the server side and JavaScript on the client side. In markets as diverse as financial software and cell phone applications, Java and .NET are the dominant choices. Until recently the performance overheads associated with managed languages were made tolerable by an exponential growth in sequential hardware performance. Unfortunately, this source of mitigation is drying up just as managed languages are becoming ubiquitous.

The hardware and software communities are thus both facing significant change and major challenges. We use a hardware-software cooperative approach to address some of these problems.

On the hardware side, we explore single-ISA heterogeneous asymmetric multicore processors (AMP) that architects have recently proposed to meet power constraints. AMP combines general-purpose *big* (fast, high power) cores and *small* (slow, low power) cores [12], [13]. Vendors such as ARM, Qualcomm, Texas Instruments, and Intel are building AMP systems already [5], [9], [10], [14]. By assigning the critical path to the big core and other execution to the small core, prior research shows how to improve performance, and consequently energy, on these architectures [9], [10], [11], [12], [13].

On the software side, we explore Virtual Machine (VM) services, such as the interpreter, compiler, profiler and garbage collector, which provide much of the abstraction of managed languages, and also much of their overhead. Since VM services execute together with every managed application, improvements to VM services will transparently improve all managed applications.

We show that the opportunities and challenges of AMP and managed languages are complimentary. The *yin* of VM services is overhead, but it meets the *yang* of small cores on an AMP. Meanwhile, the *yin* of AMP is exposed hardware complexity, but it meets the *yang* of abstraction in managed languages. The conjunction of AMP and VM services may therefore provide a win-win opportunity for hardware and



(a) Fraction of cycles spent in VM services on an i7 1C1T @ 3.4 GHz. See Section III for methodology.

(b) Execution time increase due to *adding* small cores on an AMD Phenom II.[1]

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

software communities now confronted with performance *and* power challenges in an increasingly complex landscape.

This paper identifies and leverages unique combinations of four software attributes for exploiting AMP: (1) parallelism, (2) asynchrony, (3) non-criticality, and (4) hardware sensitivity. This paper goes beyond improving performance on AMP by also seeking to improve power, energy, and PPE.

We show that VM services are *a lucrative target* 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 Oracle's HotSpot JDK actually show an even higher average fraction of time spent on garbage collection [15]. VM services in less mature VMs, such as JavaScript and PHP VMs, likely consume an even higher fraction of total cycles. Reducing the power and PPE of VM services is thus a promising target.

Figure 1(b) shows that naively executing managed applications on AMP platforms without any VM or operating system support is a very bad idea. In this figure, we measure the effect of adding small (slow, low power) cores to the performance of Java applications in Jikes RVM. The downward pointing grey arrow indicates that lower is better on this graph. The big (fast, high power) cores are out-of-order x86 cores running at the default 2.8 GHz and the small ones are simply down-clocked to 0.8 GHz on an AMD Phenom II. We evaluate both the addition of one slow to one fast, and two slow to two fast. The results are similar. Even though these configurations provide strictly more hardware resources, they slow down applications by 35% and 50% on average!

Using hardware and software configuration, power measurements, and modeling, this paper shows how to exploit the characteristics of garbage collector (GC), interpreter, and just-in-time optimizing compiler (JIT) workloads on AMP hardware to improve total power, performance, energy, and PPE. Each VM component has a unique combination of (1) parallelism, (2) asynchrony, (3) non-criticality and (4) hardware sensitivity.

Garbage Collection. Because garbage collection (GC) may be performed asynchronously and is not on the critical path, it is amenable to executing on a separate core. The computation the collector performs, a graph traversal, is parallel and thus benefits from more than one separate cores. Furthermore, GC is memory bound and many high-performance, power-hungry hardware features that improve application PPE are inefficient for GC. GC does not benefit from a high clock rate or instruction level parallelism (ILP), but it does benefit from memory bandwidth. Consequently, adding in-order low power cores with high memory bandwidth for GC does not slow GC performance much, or at all, given the right design. The result is improvement to system PPE.

JIT. The JIT is also asynchronous, exhibits some parallelism, and because its role is optimization, is generally non-critical, which also makes it amenable to executing on separate cores. The JIT workload itself is very similar to Java application workloads and PPE therefore benefits from big core power-hungry features, such as out-of-order execution, bandwidth, and large caches. We show that because the JIT is not on the critical path, executing it at the highest performance is not important. Furthermore, putting the JIT on the small core takes it off the application's critical path. Slowing the JIT down on a small core does not matter because the JIT can deliver optimized code fast enough at much less power. On a small core, we can make the JIT more aggressive, such that it elects to optimize code earlier and delivers more optimized application code, with little power cost. The resulting system design improves total performance, energy, and PPE.

Interpreter. The interpreter is, however, on the critical path and is not asynchronous. The interpreter parallelism reflects the applications' parallelism and can thus often benefit from multiple cores. However, we find that the interpreter has very low ILP, a small cache footprint, and does not use much memory bandwidth. Therefore, executing interpreter threads on a high performance, high power core is inefficient, and a better choice for PPE and energy is to execute the interpreter on low power cores.

[1]The avrora benchmark is excluded from Figure 1(b) due to its erratic behavior in this heterogeneous environment.

Each service offers a different combination of software characteristics, yet we show how to design software and hardware to improve PPE through improvements to power and performance. Using power and performance measurements of modern hardware and very conservative models, we show that the addition of small cores for GC and JIT services alone should deliver, at least, improvements in performance of 13%, energy of 7%, and performance per energy of 22%. This paper shows that (1) VM services are significant consumers of resources, (2) AMP is ineffective without hardware/software co-design, and (3) optimizing VM services and AMP hardware together should deliver substantially better total energy and PPE.

II. VM Background

We start with some background on modern virtual machine (VM) services for managed languages. Managed languages use 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. Figure 2 shows the basic VM structure.

Garbage Collection. Managed programming languages use garbage collection (GC) to provide memory safety to applications. Programmers allocate memory and the GC automatically reclaims it when it becomes unreachable. GC algorithms are graph traversal problems, amenable to parallelization, and fundamentally memory-bound. We focus on *concurrent* and parallel GC. The term concurrent denotes that the GC executes concurrently to the application and synchronizes only occasionally, while the term parallel denotes that the work of the GC is conducted by multiple GC threads. GC is prima facie an excellent candidate for parallel heterogeneous hardware.

Interpretation. Many managed languages support dynamic loading and do not perform ahead-of-time compilation. The language runtime must consequently execute code immediately, as it is loaded. Modern VMs use interpretation, template compilation to machine code, or simple compilation without optimizations (all of which we refer to as interpretation for convenience). The interpreter is thus highly responsive but offers poor code quality. Advanced VMs will typically identify frequently executed code and dynamically optimize it using an optimizing compiler. At steady state, performance-critical code is optimized and the remaining code executes via the interpreter. One exception is the .NET framework for C#, which compiles all code with many optimizations immediately, only once at, load time. As Figure 1(a) shows, about 20% of all cycles executed by the application are due to unoptimized code, amounting to about 15% of all cycles. The interpreter itself is not parallel, but it will reflect any parallelism in the application it executes. We show that the interpreter is strikingly atypical in its microarchitectural requirements, and is significantly better suited to execution on a simple core than is optimized application code.

Just In Time Compilation. High performance VMs use a JIT optimizing compiler to produce dynamically optimized code for frequently executed methods and/or traces. Because the code will have already executed at the time the optimizing JIT compiles it, the runtime has the opportunity to dynamically profile the code and tailor optimizations accordingly. A typical JIT will have several optimization levels and will select among them based on a cost model [16]. The JIT will compile code at a higher level of optimization if the predicted cost to compile and the reduction in code's execution time will reduce total time. The JIT will compile code asynchronously with the application and may compile more than one method or trace at once. JIT compilation is therefore a natural candidate for exploiting small cores.

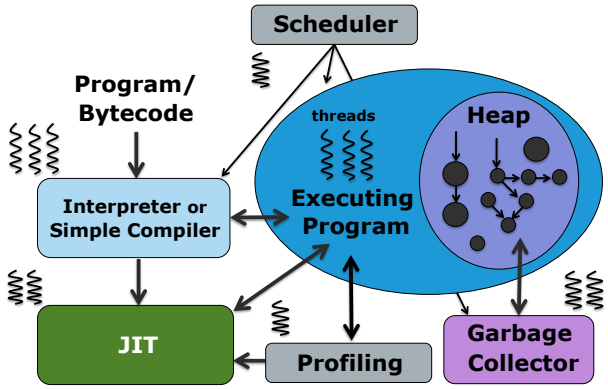


Fig. 2. Basic Virtual Machine services and structure

Other VM Services. Other VM services may present good opportunities for heterogeneous multicore architectures, but are beyond the scope of this paper. These services include zero-initialization [17], finalization, and profiling. For example, feedback-directed-optimization (FDO) 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 [18]. The profiler is parallel and exhibits an atypical memory-bound execution profile, making it a likely candidate for heterogeneous multicore architecture. However, we do not explore profiling here.

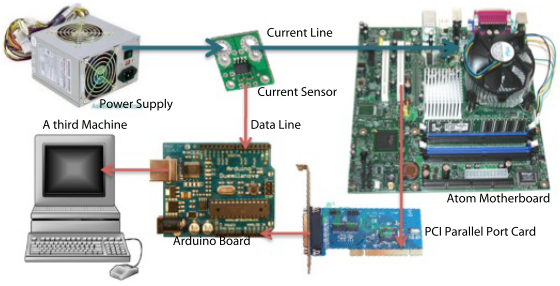
III. Experimental Methodology

We base our study on real power and performance measures of existing hardware, leveraging our existing well-developed tools and methodology [19], [20]. Evaluating the power and performance of future AMP systems is complex, just as evaluating managed languages can be challenging [20]. This section describes the hardware, measurements, workload, and software configuration that we use to explore hardware and software energy efficiency. We have made all of our data publicly available online.[2] This data includes quantitative measures of experimental error and evaluations that we could not report here due to space constraints.

[2]http://cecs.anu.edu.au/~steveb/downloads/results/yinyang-isca-2012.zip

	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	32 nm	32 nm	45 nm	45 nm
CMP & SMT	4C2T	2C2T	2C2T	6C1T
LLC	8 MB	3 MB	1 MB	6 MB
Frequency	3.4 GHz	3.3 GHz	1.66 GHz	2.8 GHz
Transistor No	995 M	504 M	176 M	904 M
TDP	95 W	65 W	13 W	125 W
DRAM Model	DDR3-1333	DDR3-1333	DDR2-800	DDR3-1333

(a) Experimental processors used in this paper



(b) Hall effect sensor and PCI card on the Atom

Fig. 3. Hardware and Power Measurement Methodology

A. Hardware

Figure 3(a) lists characteristics of the four experimental machines we use in this study. Hardware parallelism is indicated in the CMP & SMT row, and throughout the rest of this paper, 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, and 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 [21]. We use the AMD Phenom II since it exposes independent clocking of cores to software, whereas the Intel hardware does not.

Together we use this hardware to mimic, understand, measure, and model AMP designs that combine *big* (fast, high power) cores with *small* (slow, low power) cores in order to meet power constraints and provide energy efficiency architectures.

B. Power and Energy Measurement

We use on-chip energy counters provided on Intel's Sandy Bridge processors [21], and an improvement of the Hall effect sensor methodology that we previously introduced [19]. Intel recently introduced user-accessible energy counters as part of a new hardware feature called RAPL (Runtime Average Power Limit). The system 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 1 msec, and overflow about every 60 seconds [22]. Reading the MSR, we obtain package, core, and uncore energy. Key limitations of RAPL are: (1) it is only available on processors with the Sandy Bridge microarchitecture, and (2) it has a temporal resolution of just 1 msec, so it cannot resolve short-lived events. Unfortunately, VM services such as the GC, JIT and interpreter often occur in phases of less than 1 msec.

We extend the prior methodology for measuring power with the Hall effect sensor by raising the sample rate from 50 Hz to 5 KHz and using a PCI card to identify execution phases. Figure 3(b) shows a Pololu ACS714 Hall effect linear current sensor positioned between the power supply and the voltage regulator supplying the chip. We read the output using an Arduino board with an AVR microcontroller. We connect a PCI card to the measured system and the digital input of a Arduino board and use the PCI bus to send signals from the measured system to the microcontroller to mark the start and end of each execution phase of a VM service. Using the PCI card allows us to demarcate execution phases at a resolution of 200 μsec or better so we can attribute each power sample to the application or to the VM service being measured.

One limitation of this method is that the Hall effect sensor measures 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% on average, ranging form 3% to 7%. We adjust for the voltage regulator by subtracting 5% from Hall effect sensor measurements. We were unsuccessful in using this higher sample rate methodology on the AMD Phenom II, so we are limited to the lower sample rate methodology for it.

When measuring the i3 and i7, we use RAPL to measure energy and divide the measurement by time to present average power. Conversely, when measuring the Atom and AMD, we use the Hall effect sensor to take power samples and integrate them with respect to time to present energy.

Because the interpreter normally executes in phases that are shorter than any of our methodologies can resolve, we only directly measure the interpreter by running the JVM in interpreter-only mode.

Static Power Estimates: We take care to account for the static power consumption of cores when we model AMP systems using real hardware. Unfortunately, per-core static power data for production processors is not generally available, nor easy to measure [23]. By measuring power with cores in different idle states, we estimate the per-core static power on the Phenom II at 3.1 W per core. For the AtomD, we make a conservative estimate of 0.5 W per core. In the absence of better quality data, we were conservative and were able to demonstrate that our results are quite robust with respect to these estimates. The evaluation in Sections IV-B and IV-F uses the six-core Phenom II to model a single core system and a two core AMP system. Because the Phenom II powers the unused cores, we subtract the static power contribution of unused cores. When we model two Atom cores and one

Phenom II core as part of the AMP system in Section IV-F, we subtract the static power for five AMD cores and add the static power for two Atom cores.

C. Hardware configuration methodology

We use hardware configuration to explore the amenability of future AMP hardware to managed languages. We are unaware of any publicly available simulators that provide the fine-grained power and performance measurements necessary for optimizing application software together with hardware. Compared to simulation, hardware configuration has the disadvantage that we can explore fewer hardware parameters and designs. Hardware configuration has an enormous execution time advantage; it is orders of magnitude faster than simulation. In practice, time is limited and consequently, we explore more software configurations using actual hardware. Measuring real hardware greatly reduces, but does not completely eliminate, the effect of inaccuracies due to modelling.

Small Core Evaluation: To understand the amenability of the various services to small cores in an AMP design, we use the i3 and AtomD processors. The processors differ in two ways that are inconsistent with a single-die setting: a) they have different process technologies (32 nm v 45 nm), and b) they have different memory speeds (1.33 GHz vs 800 MHz). 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. Our prior work found that a die shrink from 45 nm to 32 nm reduces processor energy and power by 45% on two Intel architectures [19]. We use the same factor, but do not adjust 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 GC and JIT on a low power core, we use the Phenom II and the Hall effect sensor without the PCI card. This methodology's inability to measure fine grain events is inconsequential because we are measuring overall system power and performance in the experiments where the Phenom II is used. As mentioned above, the Phenom II's separately clocked cores make it a good base case.

Microarchitectural Characterization: We evaluate microarchitectural features using BIOS configuration. We explore the effect of *frequency scaling* on the i7, varying the clock from 1.6 GHz to 3.4 GHz. We normalize to 1.6 GHz. We explore the effect of *hardware parallelism* on the Phenom II by varying the number of participating CMP cores and on the i7 by varying CMP and SMT. We explore the effect of *last level cache* sizes by comparing the i7 and i3, each configured to use two cores at 3.4 GHz, but with 8 MB and 3 MB of LLC respectively. To explore sensitivity to *memory bandwidth*, we use the i3 with 800 MHz single channel memory relative to the default 1.33 GHz dual channel memory. 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 32 nm [19].

D. Workload

We use ten widely used Java benchmarks taken from the DaCapo suites and SPECjbb: bloat, eclipse, and fop (DaCapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (DaCapo-9.12); and pjbb2005 [20]. All are multithreaded except for fop, luindex and bloat. These benchmarks are non-trivial real-world open source Java programs [20].

E. Virtual machine configuration

All our measurements follow Blackburn et al.'s best practices for Java performance analysis [20] with the following adaptations to deal with the limitations of the power and energy measurement tools at our disposal.

Garbage Collection: We focus our evaluation on concurrent garbage collection because it executes concurrently with respect to the application and is thus particularly amenable to AMP. We also evaluate Jikes RVM's default production garbage collector, generational Immix [24]. We evaluated, but do not report, four other garbage collectors to better understand GC workloads in the context of AMP. The measurements that we do not explicitly report here are available in our online data. All of our evaluations are performed within Jikes RVM's memory management framework, MMTk [25].

We report time for the production collector in Figure 1(a) because it is the best performing collector and thus yields the lowest time. In all other cases, we use the concurrent collector. Our concurrent mark-sweep 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. We use 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 8 MB of allocation for avrora, fop, and luindex, which have a low rate of allocation, and 128 MB for the remaining benchmarks.

Measurement of concurrent GC faces two major challenges: a) we have no way of measuring the power and/or energy of a particular thread, and thus we cannot directly measure power or energy of concurrent collection, and b) unlike full heap stop-the-world (STW) collectors, which suspend all application threads when collecting, concurrent collectors require the application to perform modest housekeeping work which can not be directly measured because it is finely entangled within the application. We use a full-heap STW collector and a STW variation on our concurrent collector to solve this methodological challenge. Using the STW-concurrent collector, we can isolate and measure concurrent GC power and performance. Using the full-heap STW collector we can identify application power and time in isolation from any GC. We can then measure the net overhead of concurrent GC by subtracting application power and time from total power and time measured when using concurrent GC.

The GC implementations expose software parallelism and exploit all available hardware contexts.

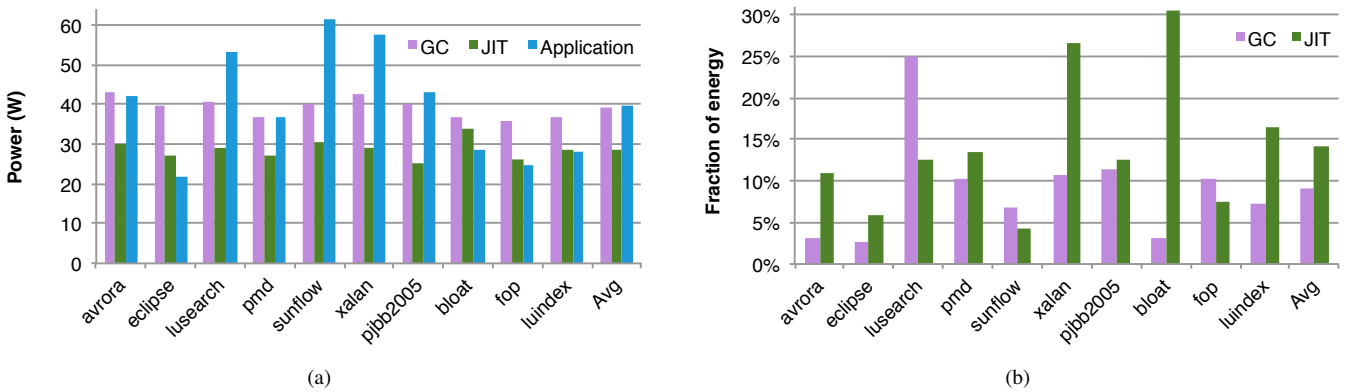


Fig. 4. GC, JIT, and application power and energy on i7 4C2T at 3.4 GHz. (a) Average power for GC, JIT, and application. (b) Fraction of energy due to GC and JIT. The power demands of the GC and JIT are relatively uniform across benchmarks. Together they contribute about 20% to total energy.

JIT Compiler: Because the unit of work for the JIT when executing normally is too fine grained for either RAPL or Hall effect measurements, we perform and measure all JIT work at once from a *replay* profile [20]. *Replay compilation* removes the nondeterminism of the adaptive optimization system. The methodology uses a compilation profile produced on a previous execution that records what the adaptive compiler chose to do. It first executes one iteration of the benchmark without any compilation, forcing all classes to be loaded. Then the compilation profile is applied en mass, invoking the JIT once for each method identified in the profile. We measure the JIT as it compiles all of the methods in the profile. We then disable any further JIT compilation and execute 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 with 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 four times the minimum size required.

Normally the JIT executes asynchronously to the application and GC. Although the JIT compiler could readily exploit parallelism by compiling different code in multiple threads, Jikes RVM's JIT is not parallel. We thus evaluate the JIT on a single hardware context. When we evaluate the JIT and GC together we use multiple hardware contexts.

Interpreter: Evaluating the interpreter is challenging because interpretation is finely interwoven with optimized code execution. We evaluate the interpreter two ways, using two JVMs, 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. We first use Jikes RVM and a timer-based sampler to estimate the fraction of time spent in interpreted (template-compiled) code. We use HotSpot to evaluate the interpreter's microarchitectural sensitivity. We execute HotSpot with the compiler turned off, so that all application code is interpreted. Because the interpreter is tightly coupled with the application, it exhibits no independent parallelism and cannot execute asynchronously with respect to the application. However, it reflects all software parallelism inherent in the application.

IV. Experimental Analysis

We now present a quantitative analysis of garbage collection, JIT compilation, and interpretation with respect to their suitability to asymmetric multicore processors (AMP). We start with a motivating analysis of the power and energy footprint of VM services on orthodox hardware. We then assess the amenability of VM services to a small core. We evaluate the extent to which VM services can benefit from targeted microarchitectural optimizations, an opportunity that AMP offers. Finally we model the overall impact of executing VM services on a simple AMP system.

A. Motivation: Power and Energy Footprint of VM Services

Figure 4 shows the overall contribution of the GC and JIT to system power and energy on existing hardware. As we mentioned in the previous section, the interpreter's fine-grained entanglement with the application makes it too difficult to isolate and measure its power or energy with current tools. Figure 1(a) shows that the fraction of cycles due to the interpreter is significant. Although we do not evaluate the interpreter further here, it remains a significant target for power and energy optimization on future AMP systems. Figure 4 reports isolated GC and JIT power and energy on a stock i7 (4C2T at 3.4 GHz). Our methodology generates typical JIT and GC workloads. We use the JIT workload during warm-up (the first iteration of each benchmark), in which it performs most of its work. We use the GC and application workload in steady state (5th iteration), where the GC sees a typical load from the applications and the application spends most its time in optimized code. Since the GC is concurrent and parallel, it utilizes all available hardware. The JIT is concurrent, but single threaded, although JIT compilation is not intrinsically single threaded. The benchmarks themselves exhibit varying degrees of parallelism.

Figure 4(a) shows that power consumption is quite uniform for the GC and JIT regardless of benchmark on the i7, at around 40 W and 30 W respectively. The JIT has lower power because it is single threaded, whereas the parallel GC uses all

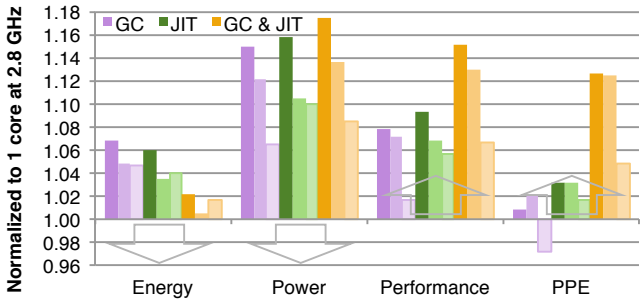


Fig. 5. Utility of adding a core dedicated to VM services on total energy, power, time, and PPE. Overall effect of binding GC, JIT, and both GC & JIT to the second core running at 2.8 GHz (dark), 2.2 GHz (middle), and 0.8 GHz (light) on the AMD Phenom II corrected for static power. The baseline uses one 2.8 GHz core.

four cores and SMT. This power uniformity shows that the GC and JIT workload are both largely benchmark independent. By contrast, application power varies by nearly a factor of three, from 20 W to 60 W, reflecting the diverse requirements and degrees of parallelism among the benchmarks.

Figure 4(b) shows that the GC and JIT each contribute about 10% on average to the total energy consumption, totalling about 20%. This confirms our hypothesis that VM services may provide a significant opportunity for energy optimization. The figure also shows significant variation in GC and JIT energy consumption across benchmarks, despite their uniform power consumption. This variation reflects the different *extents* to which the benchmarks exercise the JIT and GC. Some benchmarks require more or less GC and JIT services, even though the behavior of each service is quite homogeneous.

B. Amenability of VM Services to a Dedicated Core

The above results demonstrate the potential for energy savings and we now explore the amenability of executing VM services on dedicated small cores. This experiment explores whether the services will a) benefit from dedicated parallel hardware, and b) be effective, even if the hardware is slow.

Because we are not yet considering the small core microarchitecture, we use existing stock hardware. We choose the AMD Phenom II because it provides independent frequency scaling of the cores. We model the small core by down-clocking a regular core from 2.8 GHz to 2.2 GHz and 0.8 GHz. We subtract our conservative estimate of static power 3.1 W per core from our measurements to avoid overstating our results. (Section III-B explains the estimation.)

We bind the VM service(s) to separate cores and measure the entire system. We use regular adaptive JIT compilation because it interleaves its work asynchronously and in parallel with the application (replay compilation, although easier to measure, does neither). We measure total time for the first iteration of each benchmark because the first iteration is a representative JIT workload that imposes significant JIT activity (see Figure 1(a)).

Figure 5 shows the effect of adding a dedicated core for GC and JIT services on energy, power, performance, and PPE. The light grey arrows show which direction is better (lower for energy and power, and higher for performance and PPE). The leftmost bar in each cluster shows the effect when the additional core runs at 2.8 GHz, the same speed as the main core. This result evaluates our first question: whether the system benefits from hardware parallelism dedicated to VM services. For both the GC and JIT, the introduction of dedicated hardware improves performance by 8-10% while increasing power by around 15%. They independently increase energy by around 6% and marginally improve PPE. When both GC and JIT are bound to the additional core the effect is amplified, leading to a net PPE improvement of 13% and a 2% increase in energy. This data shows that simple binding of the JIT and GC to an additional core is effective.

In this paper however, we are exploring future systems in which adding more big cores is not feasible because of power or energy constraints. If it were, dedicating a big core to the VM services rather than sharing it with the application is probably a poor design choice.

The lighter bars in Figure 5 show the effect of slowing down the dedicated VM services core. The power overhead of the additional core is reduced by nearly half when the dedicated core is slowed down to 0.8 GHz. However performance also suffers, particularly for GC. Nonetheless, the PPE improves for the JIT and JIT & GC cases, which indicates that the GC and JIT could efficiently utilize a small core. A very promising result from Figure 5 is that executing JIT and GC on the dedicated core at 2.2 GHz delivers a 13% higher PPE than one processor with virtually no energy cost.

We also measure the effect on power and energy of introducing a small core *without* binding the VM services. This configuration led to a significant slow down, which is illustrated in Figure 1(b). There was also a modest increase in energy, which together lead to a 30% degradation in PPE. This result emphasizes that without binding of tasks or some other guidance to the OS scheduler, the addition of a small core is counterproductive.

C. Amenability of VM Services to Hardware Specialization

A single-ISA heterogeneous processor design has the opportunity to include a variety of general purpose cores tailored to various distinct, yet ubiquitous, workload types. Such a design offers an opportunity for very efficient execution of workloads that do not well utilize big out-of-order designs. We now explore the amenability of VM services to alternative core designs. We start with measuring a stock small processor, the in-order, low power, AtomD. We then evaluate how the VM services respond to a range of microarchitectural variables such as clock speed, cache size, etc. To ease measurement, this analysis evaluates the GC and application with respect to a steady state workload and the replay JIT workload. We evaluate the interpreter using the HotSpot JDK with its JIT disabled and measure the second iteration of the benchmark, which reflects steady state for the interpreter.

Small Core: We first compare contemporary in-order cores to out-of-order cores. We use an i3 and AtomD with the

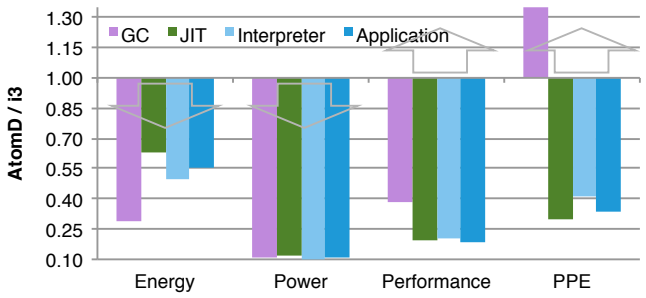


Fig. 6. Amenability of services and the application to an in-order-processor. These results compare execution on an in-order AtomD to an out-of-order i3.

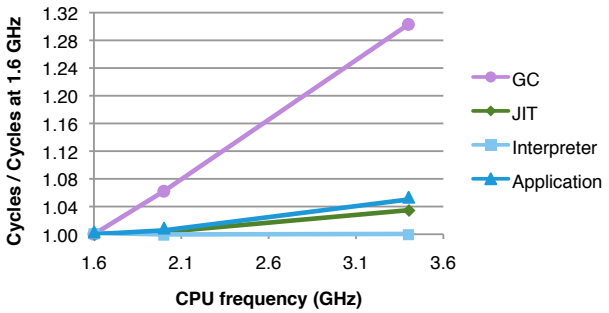


Fig. 7. Cycles executed as a function of clock speed, normalized to cycles at 1.6 GHz on the i7. The workload is fixed, so extra cycles are due to stalls.

same degree of hardware parallelism (2C2T), and run with the same memory bandwidth to focus on the microarchitectural differences. We do not adjust for clock frequency in this experiment on the grounds that the small core may well run at a lower clock, so they execute at their 3.4 GHz and 1.66 GHz default frequencies respectively. We explore the effect of clock scaling separately, below. An important difference between the two machines is their process technology. To estimate the effect of shrinking the process, we project the AtomD power consumption data to 32 nm by reducing measured power by 45%, as described in Section III-C.

Figure 6 shows the effect on energy, power, performance and PPE when moving from the i3 to the in-order AtomD. The figure shows, unsurprisingly, that the Atom offers lower energy, power, and performance in all cases. Power is uniformly reduced by around 90%. Of these, the GC benefits the most because its performance degradation is less than for the others. The consequence is greater energy reduction and a net improvement in PPE of 35%. The JIT, interpreter and application all see degradations in PPE of around 60-70%.

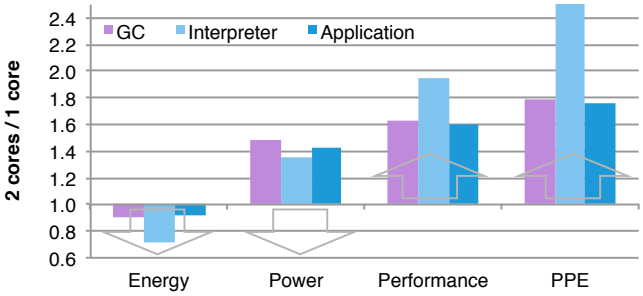
This data makes it emphatically clear that the simple memory-bound graph traversal at the heart GC is much better suited to small low power in-order processors. In the case of the JIT, the 35% energy reductions may trump degraded PPE because the JIT can be trivially parallelized, which improves performance without significantly increasing energy, leading to improved PPE. The small core will be 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.

Figure 7 provides insight into why the GC does so well on the Atom. This graph shows the effect on the total cycles executed when scaling the clock on a stock i7. The application and JIT execute just 4-5% more cycles as the clock increases from 1.6 GHz to 3.4 GHz while the GC executes 30% more cycles. These extra cycles are due to stalls. The result is unsurprising given the memory-bound nature of garbage collection. It is also interesting to note what this graph reveals about the interpreter. The higher clock speed induces no new stalls, so the interpreter's performance scales perfectly with the clock frequency. This result reflects the fact that an interpreter workload will invariably have very good instruction locality because of the interpreter loop that dominates execution. We confirmed this by directly measuring the frequency of last level cache accesses by the interpreter and found that it was 70% lower than the application code. Both the interpreter and the GC exhibit atypical behavior that suitably tuned cores should be able to aggressively exploit for greater energy efficiency.

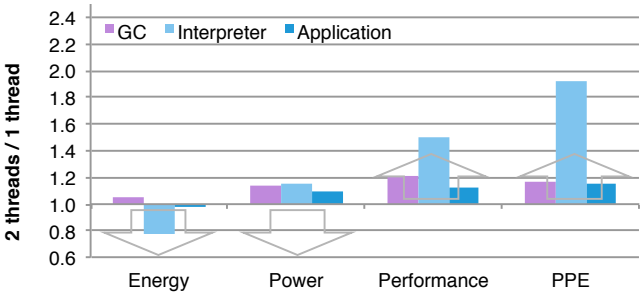
Microarchitectural Characterization: We explore the sensitivity of the GC, JIT, interpreter, and application workloads to microarchitectural characteristics. This characterization gives further insight into the amenability of the services to hardware specialization. Figure 8 shows the result of varying: hardware parallelism, clock speed, memory bandwidth, cache size, and gross microarchitecture. For the SMT and CMP experiments, we drop the three single threaded benchmarks, and use the seven multithreaded ones to focus on sensitivity with software parallelism. Similarly, six of the applications fit in the smaller 3 MB last-level cache, so we use the ones that do not (bloat, eclipse, pmd and pjbb) to compare with VM services. To generate the GC workload, we drop avrora, fop, and luindex here because they have high variation and low GC time. Note the different y-axis scales on each graph.

Hardware parallelism. We study the effects of hardware parallelism using the i7. To evaluate CMP performance, we compare 1C1T and 2C1T configurations, which disable hyperthreading and ensure that software parallelism is maximally exposed to the availability of another core. Conversely, to evaluate SMT performance, we compare 1C1T and 1C2T configurations, which use just one core to maximize exposure of software parallelism to the addition of SMT. In all cases, the processor executes at its 3.4 GHz stock frequency. Because our JIT is single threaded, we omit the JIT from this part of the study. However, some JITs are parallel and would be amenable to hardware parallelism.

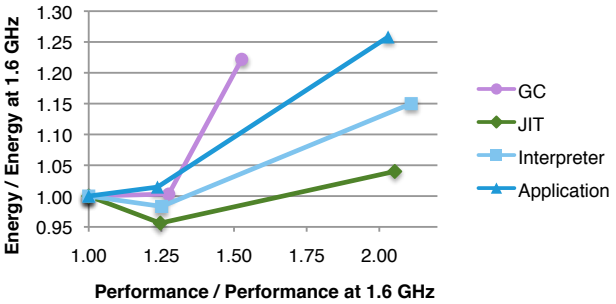
Figure 8(a) shows that increasing the number of cores improves both GC and multithreaded applications' PPE similarly. The interpreter sees an even greater benefit, which is likely due to the lower rate of dependencies inherent in the slower interpreter loop relative to typical optimized application code. Figure 8(b) shows that SMT does not improve PPE as



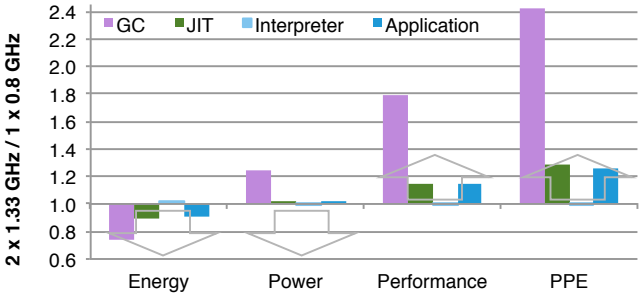
(a) Effect of CMP on GC, interpreter, and application.



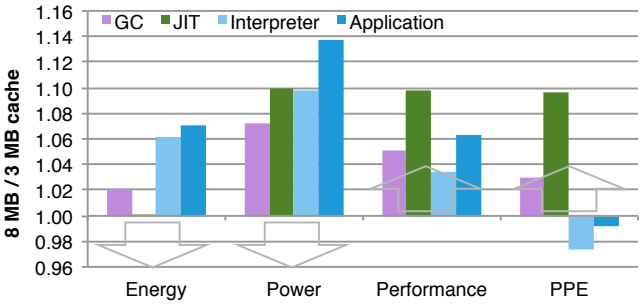
(b) Effect of SMT on GC, interpreter, and application.



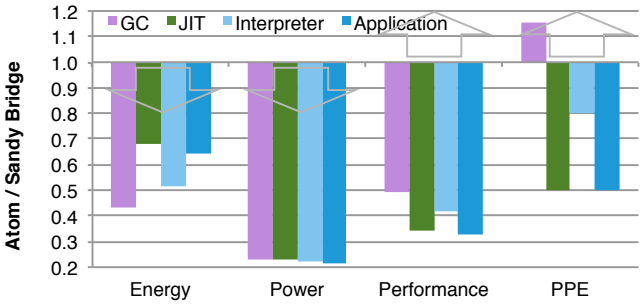
(c) Effect of clock frequency on performance and energy. Points are clock speeds 1.6, 2.0 and 3.4 GHz (from left to right) on i7 4C2T.



(d) Memory bandwidth effect. The high memory bandwidth is 1.33 GHz x 2 channels relative to 0.8 GHz x 1 channel.



(e) LL cache size effect on i7 and i3 with 8 MB and 3 MB LL cache respectively.



(f) Gross architecture effect on Sandy Bridge and Atom at same frequency, hardware threads, and process technology.

Fig. 8. Microarchitectural characterization of VM services and application.

much as CMP, but effectively decreases energy. SMT requires very little additional power as compared to CMP and is a very effective choice given a limited power budget [19]. The advantage of the interpreter over the other workloads is even more striking here. Six multithreaded benchmarks exhibit dramatic interpreter performance improvements on SMT and CMP respectively: sunflow (87%, 164%), xalan (54%, 119%), avrora (55%, 74%), pmd (45%, 113%), lusearch (53%, 111%), and pjbb (36% and 67%). Eclipse is the only multithreaded benchmark that did not improve due to hardware parallelism when fully interpreted. These results show that VM services can very effectively utilize available hardware parallelism.

Clock speed. Figure 8(c) plots performance (x-axis) and energy (y-axis) as a function of clock frequency on the i7. The single threaded JIT uses a 1C1T configuration, while all other workloads use the stock 4C2T configuration. Values are normalized to those measured at the minimum clock frequency (1.6 GHz). The different responses of each workload is striking. The JIT, interpreter, and application all improve their performance two-fold as the clock increases. On the other hand, the GC performance improves very little beyond 2.0 GHz. Figure 7 and our measurements of cache miss rates suggest that memory stalls are the key factor in this result. The different workloads also have markedly different energy responses. The single-threaded JIT energy only increases by 4% going from the lowest to the maximum clock speed, whereas the application and GC energy consumption increase by 25% and 23% respectively with clock speed increases.

Memory bandwidth. Figure 8(d) shows the effect of increasing memory bandwidth from a single channel at 0.8 GHz to

two channels at 1.33 GHz. The increase in memory bandwidth reduces CPU energy for all workloads, but most strikingly for GC. The GC performance increases dramatically, leading to a 2.4× improvement in PPE. Of course the increased memory bandwidth will lead to increased energy consumption by the off-chip memory subsystem, and our measurements are limited to the processor chip and thus not captured by this data. The GC's response to memory bandwidth is unsurprising, given that the workload is dominated by a graph traversal. It is interesting to note that while the JIT and the application also see significant, if less dramatic, improvements, the interpreter does not. The results in Figure 8(d) are quite consistent with the results in Figure 7. The GC is memory-bound and sensitive to memory performance. The interpreter has excellent locality and is relatively insensitive to memory performance.

Last-level cache size. A popular use of abundant transistors 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 (3.4 GHz). 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 8 MB and i3's is 3 MB. Six of our ten benchmarks are insensitive to the large cache size. Cache is not a good use of transistors for them. The four cache-sensitive benchmarks (bloat, eclipse, pmd and pjbb2005) have large minimum heap sizes of more than 100 MB. For these benchmarks, the increase in LL cache size from 3 MB to 8 MB is approximately 3% of the average maximum volume of live objects. Note the y-axis scale in Figure 8(e). Even these benchmarks only improve performance by 6% with a larger cache. The larger cache improves JIT performance and PPE by 10%, but these improvements are very modest in comparison with the other hardware features explored in Figure 8. The interpreter, which has good locality, sees a reduction in PPE when the cache size is increased.

Gross microarchitecture. Figure 8(f) compares the impact of gross microarchitecture using the AtomD and i3, controlling for clock frequency (1.66 GHz for AtomD and 1.6 GHz for i3), hardware parallelism (2C2T), and memory bandwidth. We configure them both to use 800 MHz single channel memory. We adjust for process technology on the AtomD, as we do above, by scaling by 45%. The results make it clear that the GC is better suited to the small in-order AtomD than a big core, with the high performance architectural features of the i3. GC has a better PPE on the AtomD than on the i3. The interpreter does not do as well as the GC, but the benefit of the i3 is muted compared to that for the JIT and the application.

D. Discussion

These results show that each of the VM services exhibit a distinctly different response to microarchitectural features. The similarities between the JIT and application are not a surprise. The characteristics of compilers have been studied extensively in the literature and are included in many benchmark suites including the ones we use here. On the other hand, the GC and interpreter each exhibit striking deviations from the application. This heterogeneity presents designers with a significant opportunity for energy optimizations that tune small general purpose cores more closely to the microarchitectural needs of this ubiquitous workload.

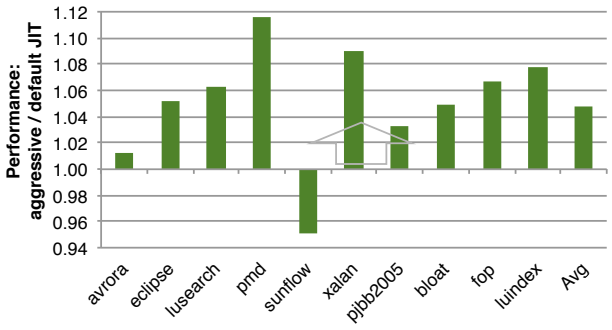


Fig. 9. Adjusting the JIT's cost-benefit model to account for lower JIT execution costs yields improvements in total application performance.

E. Further Opportunities for the JIT

Although the JIT is broadly similar to the applications and therefore may not offer an opportunity for tuned hardware, executing the JIT on small cores does offer 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 9 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 analysis (see Section II), to a more aggressive cost model model that compiles code sooner and compiles more code. We configure the compiler cost model by reducing the cost of compilation by a factor of 10. Figure 9 shows that making the JIT more aggressive improves total performance by around 5%. This result suggests that software/hardware co-design has the potential to improve power, performance, and PPE further.

F. Modeling Future AMP Processors

This section evaluates a hypothetical AMP scenario. The absence of a suitable AMP processor in silicon and our software constraints for executing on an x86 ISA motivate using a model. Our model combines measured application power and performance results on the big Phenom II core at 2.6 GHz with the power and performance results for GC and JIT on two small Atom cores at 1.66 GHz. We compare this model to the Phenom II AMP core results from Figure 5, which we corrected for static power. Because the Atom cores are projected into the Phenom II die, we assume the Phenom II's memory bandwidth and LL cache size, and accordingly raise the Atom's performance by 79% and 5% for the GC and 2% and 10% for the JIT, based on our measurements in Figures 8(d) and 8(e).

For the small cores, which run the VM services, we model GC throughput by straightforwardly scaling up our measures

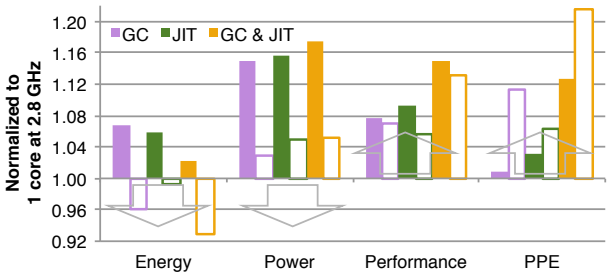


Fig. 10. Modeling total energy, power, time, and PPE of an AMP system. The light bars show a model of an AMP with one 2.8 GHz Phenom II core and two Atom cores dedicated to VM services. The dark bars reproduce Phenom II results from Figure 5 that use a 2.8 GHz dedicated core for VM services.

of GC throughput on the regular Atom according to the adjustments for the Phenom II cache size and memory bandwidth. We do the same for the JIT, however, we also scale the JIT from 1C1T to 2C2T making the assumption that the JIT will scale as well as typical applications. This adjustment is conservative since the JIT is embarrassingly parallel.

To make our model more accurate, we capture the effect on the application of slowing down the GC and JIT. We leverage the results on the Phenom II that down-clock cores in Figure 5. The modeled performance for the GC on the two Atom cores is slightly better than on the Phenom II at 2.2 GHz, and the JIT modeled on the two Atom cores is slightly better than on the Phenom II at 0.8 GHz. We thus use the application measurements when running with the 2.2 GHz and 0.8 GHz dedicated VM services cores respectively. This measurement conservatively captures the effect on the application of slowing the GC and JIT.

Figure 10 shows that even our conservative model of the two Atom cores (light bars) is promising. For the GC alone, energy improves by 4% due to a performance improvement of 7% but at only a 3% power cost. The JIT alone has slightly worse total performance impact, but at a commensurate power cost. Individually, the PPE improvement is 11% and 6% from GC and JIT respectively. When we bind both the GC and JIT to the small cores, the result is more than additive results because of the better utilization of the small cores. Total performance is very similar to that of the 2.8 GHz dedicated core but power, energy, and PPE are all markedly improved. The performance improvement is 13% with only a 5% power cost, resulting in a 7% energy improvement and a 22% PPE improvement.

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

Several recent publications describe operating system scheduling algorithms for improving performance on AMP hardware [9], [10], [11], [12], [13], but the combination of AMP and VM services is unique to our work. Some work has however focused on architecture specialization for the VM workload. Most of this work focuses on GC, although some early LISP work considers architectures for JIT and interpretation [27].

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]. Their simulation indicate that 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 for satellites and aircraft. In comparison, we focus on general purpose hardware and software. Azul systems built a custom chip to run Java business applications [35]. They redesign about 50% of the 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. On the other hand, many application developers are choosing managed languages, which insulate them from hardware change by a layer of abstraction. One emerging hardware trend is asymmetric multicore processors (AMP) with a single ISA. This paper shows that a software/hardware co-design for big and small cores and the system software, and in particular VM services, is a promising approach. Our data suggests that small cores tailored to services such as garbage collection, the interpreter, and JIT will help AMP deliver on their energy efficiency promise. By targeting VM services, our approach has broad reach, does not burden application programmers, nor does it compromise application portability to rapidly evolving hardware.

REFERENCES

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

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

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

[4] Japanese Green IT Promotion Council, "Concept of new metrics for data center energy efficiency," 2011. [Online]. Available: http://www.greenit-pc.jp/e/topics/release/100316_e.html

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

[6] 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.

[7] 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 *ACM International Conference on Architectural Support for Programming Languages and Operation Systems*, 2010, pp. 205–218.

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

[9] 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 Symposium on High Performance Computer Architecture*, 2010, pp. 1–12.

[10] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *European Conference on Computer Systems*, 2010, pp. 125–138.

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

[12] 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," *Journal of Parallel and Distributed Computing*, vol. 71, no. 1, pp. 114–131, 2011.

[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 International Symposium on Microarchitecture*, 2003, pp. 81–92.

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

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

[16] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, "Adaptive optimization in the Jalapeño JVM (poster session)," in *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, 2000, pp. 125–126.

[17] X. Yang, S. Blackburn, D. Frampton, J. Sartor, and K. McKinley, "Why nothing matters: The impact of zeroing," in *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, 2011, pp. 307–324.

[18] J. Ha, M. Arnold, S. Blackburn, and K. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, 2009, pp. 155–174.

[19] 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 *ACM International Conference on Architectural Support for Programming Languages and Operation Systems*, 2011, pp. 319–332.

[20] 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 Conference on Object–Oriented Programming Systems, Languages, and Applications*, Oct. 2006, pp. 169–190.

[21] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le, "RAPL: Memory power estimation and capping," in *International Symposium on Low Power Electronics and Design*, 2010, pp. 189–194.

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

[23] E. Le Sueur, "An analysis of the effectiveness of energy management on modern computer processors," Master's thesis, School of Computer Science and Engineering, University of NSW, Sydney, 2011.

[24] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality," in *ACM Conference on Programming Language Design and Implementation*, 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 *ACM Conference on Measurement & Modeling Computer Systems*, 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.

[27] R. D. Greenblatt, T. F. Knight, J. T. Holloway, and D. A. Moon, "A lisp machine," in *Computer Architecture for Non-Numeric Processing*, 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 *International Symposium on High Performance Computer Architecture*, 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 *International Conference on Parallel Processing Workshops*, 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 *The ACM International Symposium on Memory Management*, 2007, pp. 173–182.

[34] ——, "Exploiting the efficiency of generational algorithms for hardware-supported real-time garbage collection," in *ACM Symposium on Applied Computing*, 2007, pp. 713–718.

[35] C. Click, "Azul's experiences with hardware/software co-design. Keynote presentation at the ACM SIGPLAN," in *ACM International Conference on Virtual Execution Environments*, 2009.