Dynamic Heterogeneity and the Need for Multicore Virtualization

Philip M. Wells
Google, Inc.
pwells@google.com

Koushik Chakraborty
Electrical and Computer Engineering
Utah State University
kchak@engineering.usu.edu

Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
sohi@cs.wisc.edu

Abstract

As the computing industry enters the multicore era, exponential growth in the number of transistors on a chip continues to present challenges and opportunities for computer architects and system designers. We examine one emerging issue in particular: that of *dynamic heterogeneity*, which can arise, even among physically homogeneous cores, from changing reliability, power, or thermal conditions, or different cache and TLB contents. This heterogeneity results in a constantly varying pool of hardware resources, which greatly complicates software's traditional task of assigning computation to cores.

In part to address dynamic heterogeneity, we argue that hardware should take a more active role in the management of its computation resources. We propose hardware techniques to virtualize the cores of a multicore processor, allowing hardware to flexibly reassign any number of the *virtual processors* that are exposed to a single operating system to any subset of the physical cores. We show that multicore virtualization operates with minimal overhead, and that it enables several novel resource management applications for improving both performance and reliability.

1. Introduction

Advances in technology are continuing to drive Moore's Law and double the number of transistors available on a chip every two years. This exponential growth, however, presents several challenges in determining how to use those transistors effectively. In past decades, computer architects were able to take these additional transistors and use them for creative purposes, such as out-of-order execution and deep speculation, transparently improving the performance of unmodified applications. However, a multitude of issues have come to prevent large, monolithic uniprocessors from continuing as a viable design, leading the way for the rapid emergence of multicore processors [10, 21]. Instead of trying to create one large, complex core to use all of the available transistors, multicore processors integrate several cores across the chip, allowing the performance of certain applications to continue to scale with Moore's Law without requiring frequent, slow, power-hungry cross-chip communication.

Though perhaps inevitable, multicore processors have presented numerous new challenges to architects, system designers, and application programmers alike. As widely noted in the research community, software (including the Hypervisor/VMM, OS, compiler, and application), must now extract enough concurrency to keep all the cores busy. This is proving to be an incredibly challenging task, but even that is not enough: In a traditional multicore processor, software must also explicitly manage the use of those cores in order to express its concurrency to the hardware.

We argue that efficiently managing the use of all on-chip cores will soon become equally as challenging for software as extracting concurrency. The reason is due to the emergence of *dynamic heterogeneity* among on-chip cores, arising from rapidly changing characteristics and requirements, such as reliability, power, or thermal conditions, or different cache and TLB contents. Such dynamic heterogeneity can exist even if the cores are physically homogenous in their design. Dynamic heterogeneity creates a constantly varying pool of resources, whose configurations and capabilities change more rapidly than software can adapt. In addition, we argue that for many types of layered systems, software should not have to understand and adapt to such low-level hardware uncertainty, even if it were able.

In the spirit of dynamically scheduled out-of-order processors, which remove software's burden of directly managing the functional units of a single core, we propose to virtualize the multiple on-chip cores, enabling hardware/firmware to flexibly map computation onto the most appropriate core at any given time. Our proposed *Multicore Virtualization* allows the hardware designer to abstract the low-level details of the cores, such as their dynamic heterogeneity, in order to alleviate software from the burden and inefficiency of managing these resources directly.

2. Dynamic Heterogeneity and Implications

Several proposals have exalted the benefits of designing processors with *statically heterogeneous* cores — cores that are designed to have different physical characteristics in order to capitalize on different engineering trade-offs (e.g., [13, 18]). Future multicore chips will similarly contain *dynamically heterogeneous* cores as well — cores which exhibit different, and rapidly changing, execution characteristics, even though they may be physically homogeneous in design.

2.1 Why Dynamic Heterogeneity?

Dynamic heterogeneity arise from a variety of sources, including varying reliability, power, or thermal constraints of the circuits, or changing contents of cache, TLB, or other microarchitectural structures. Dynamic heterogeneity can also refer to the varying configurations of hardware resources, particularly as they relate to changing software requirements such as the need for extra reliability or instruction-level parallelism. Dynamic heterogeneity is a result of two byproducts of Moore's Law: less reliable transistors, and increased complexity coming from the use of those transistors.

In the first case, as individual transistors become smaller and closer together, they begin to suffer from a number of problems. For example, smaller transistors become more susceptible to *transient*, *permanent*, and *intermittent* hardware faults [4, 5, 9, 29, 31], which can impact an individual core's ability to perform reliable computation. Components such as register files can suffer from power density problems, as power-hunger transistors in a small area struggle to dissipate the heat from their heavy utilization. Global (chip-wide) power distribution and dissipation limitations can prevent all cores from operating at peak performance at the same time. A number of techniques to tackle these problems have been proposed, such as dynamically adjusting voltage and/or frequency [6, 32, 45], or temporarily stopping the use of an affected core altogether [8, 14, 27, 42]. Though these solutions are effective in many cases at mitigating transistor issues, these solutions themselves create dynamic heterogeneity that becomes software-visible.

In the second case, as transistors become more numerous, microarchitects and circuit designers hook them up in increasingly complex ways. One source of complexity arises from predictive microarchitectural structures within each core, including branch predictors, caches, and TLB arrays. As a program executes, these structures become dynamically tailored for running a particular fragment of code efficiently. Such dynamic specialization is desirable, but it mandates an additional level of complexity to exploit its performance benefits. In Section 6, we explore two examples of how such heterogeneity can be purposefully created and exploited through careful assignment of computation to cores.

Another source of complexity arises due to the new proposed capabilities among multiple cores, including the ability to join two or more cores together for reliability or extracting concurrency. As one example, an applications may have certain components that require Dual-Modular Redundancy (DMR) (e.g., [3, 34]) to maintain sufficient levels of reliability in future systems, while other portions of the same application can tolerate higher rates of hardware faults and avoid the power and performance penalty of DMR. We discuss this example in more detail in Section 5.2. As another example, software has changing levels of parallelism. Sometimes, task- or instruction-level parallelism of a single thread can be exploited via multiple conjoined cores through speculative multithreading (e.g., [35]) or dynamic *Core Fusion* (e.g. [17]). At other times, those cores can be reconfigured to take advantage of data-level parallelism (e.g., [28]), or used to exploit thread-level parallelism by independently executing multiple threads. The dynamic coupling of cores not only creates different capabilities of each *logical* core, but also a varying *number* of logical cores, capable of concurrently executing a varying number of software threads.

The end result is that dynamic heterogeneity creates a system where frequent changes in the characteristics, configuration and number of available or appropriate cores, are the expectation, not the exception.

2.2 Software Implications

These emerging opportunities and challenges of dynamic heterogeneity share two distinct traits. First, the details create a new layer of complexity between the computation and the physical hardware performing that computation. This complexity in turn creates uncertainty in how computation should be efficiently mapped onto the hardware at any given moment. Unraveling this uncertainty requires detailed knowledge of the current configuration and capabilities of each core — information that modern system and application software does not posses, and cannot easily acquire due to the relatively static interfaces present in layered systems. Second, the capabilities and configurations of the hardware can change rapidly, yet the cost of trapping into the OS to perform and implement a scheduling decision has actually *increased* over the years, relative to the cost of computation [25]. As a result, system and application software often cannot implement policies to address dynamic heterogeneity with sufficient timeliness, even if interfaces *was* modified to provide appropriate information [42].

2.3 The Need for Multicore Virtualization

Given these two issues, we argue that hardware should take a more active role in the management of its resources, in particular, the on-chip cores. By abstracting the details of the on-chip cores using a thin multicore virtualization layer, simple homogeneous *virtual processors* (VCPUs) can be exposed to the OS via the hardware/software interface (i.e., ISA), while innovations in multicore hardware adapt to the opportunities and challenges of dynamic heterogeneity of the underlying physical cores. In this way, the lowest level of system software (e.g., the OS or traditional software VMM) can remain completely unmodified.

Figure 1 illustrates multicore virtualization and its two basic abilities. First, it supports the ability to move a software-visible VCPU from one core to another, and the

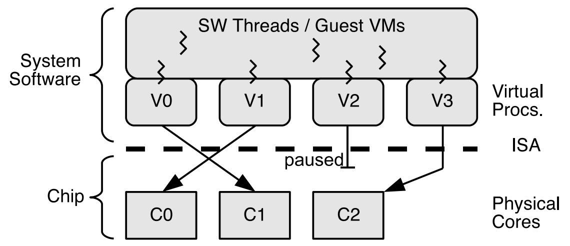

Figure 1. Multicore Virtualization. *Four VCPUs are exposed the software, only three cores are actually present. VCPUs V0, V1, and V3 have been transparently migrated, while VCPU V2 has been transparently suspended.*

ability to temporarily suspend execution of a VCPU when there are no appropriate cores on which it can run. Second, is preserves the illusion that all VCPUs are simultaneously executing, even if a subset of them are suspended.

3. Experimental Methodology

In the following sections, we present a number of experiments to examine the effectiveness of multicore virtualization, and four examples of applications which use this virtualization to adapt to, or take advantage of, dynamic heterogeneity. Our primary mode of experimentation is though full-system simulation using Virtutech Simics [23]. Simics is an execution driven simulator which functionally models a *SunFire 6800* server in sufficient detail to boot unmodified operating systems. We use Simics as a functional simulator only, and model the timing of our target multicore microarchitecture using Simics MAI and our own cycle-accurate processor and memory hierarchy module.

We model each core as having an 8-stage pipeline, with an out-of-order, 2-wide issue, a 128-entry instruction window, and operating at 3 GHz. The chip consists of 8 cores (16 for Section 5.2). Located with each core are split I&D caches, and a unified private L2. We also model a shared L3 that is exclusive with the L2s, and has a 55-cycle load to use latency. To evaluate multicore virtualization, we simulate a thin virtual-machine layer implemented primarily in hardware. We do model the overhead of maintaining VCPU state, and its effect on caches and TLBs.

We use several workloads for these experiments, all of which are running on Solaris 9. For experiments in Section 5.2 and 6.2, we use consolidated server workloads, which combine two guest VMs each running a single application. Each guest VM is configured with its own I/O devices and physical memory space, but VMs dynamically share the processors and caches. We are assuming the use of a software VMM, similar to VMWare ESX Server, which virtualizes I/O, memory, and privileged instructions. We do not model the overhead of virtualizing memory or I/O. The two guest OSs are allocated enough physical memory so that the VMM does not need to swap *real* memory.

Simulations run for 100 million to 1 billion cycles. Due to workload variability, we simulate multiple runs and report average results with 95% confidence intervals on most graphs. We use *committed user instructions* as our metric for 'work' in all experiments. Additional methodology details are provided in the papers from which these results are extracted [7, 41–43].

4. Multicore Virtualization

This section discusses our proposed techniques for multicore virtualization, which can enable hardware innovation, while remaining transparent to the system and application software. Virtualization of an entire system involves many complex tasks, especially for running multiple VMs on a single machine. Our goal, however, is much simpler: to create a framework for supporting applications which address dynamic heterogeneity. These applications suggest two main requirements, as outlined in Section 2: 1) the need for moving a VCPU from one core to another, and suspending the VCPU when there are no appropriate cores on which it can run, and 2) the need for presenting an illusion that the OS is executing on an unvirtualized multicore.

VCPU Context Switch and Migration To enable context switch, migration, and suspension of VCPUs, our proposal virtualizes the processor state, TLBs, and interrupts. To efficiently handle VCPU state, we propose a simple mechanism with limited hardware support, and no special-purpose storage, by simply storing the VCPU state in the memory hierarchy on a VCPU context switch. This task can be performed using either a hardware state machine or microcode. This proposal allows state to be transparently migrated from one core to another using the existing cache coherence protocol. The latency of a simple VCPU context switch is determined by the available memory read (and write) ports and the cache bandwidth, while a migration to another core is limited by on-chip cache-to-cache bandwidth. The state of a suspended core is placed in the caches, and can be evicted to main memory. For the UltraSPARC IIICu architecture we functionally model, this state is 2.2KB per VCPU.

SPARC V9 uses a software managed TLB, which means that most storage and operational aspects of the TLB are architected and OS-visible. TLB control registers are migrated as part of the VCPU state, but we make the observation, at least for Solaris and Linux, that most TLB entries can be shared among VCPUs. Sharing is possible because these OSs tags each entry with a context ID for each process address space, but share these context IDs across VCPUs. Sharing reduces the amount of VCPU state that much be managed, but also allows constructive and destructive interference among VCPUs.

To properly handle both hardware and software initiated interrupts, we use a centralized controller, implemented in hardware. A table is used to map an incoming interrupt from the proper VCPU to the core currently executing that VCPU.

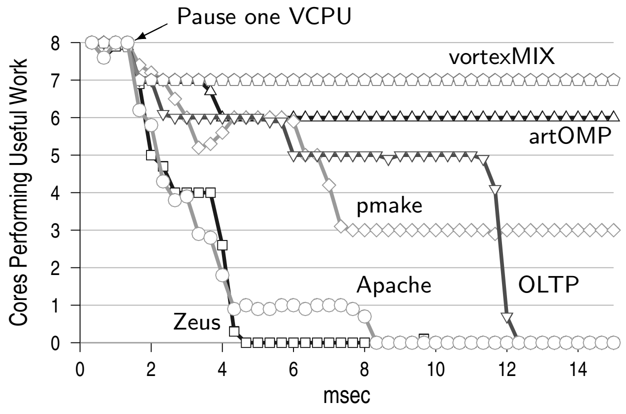

Figure 2. Livelock of Naive Overcommitting

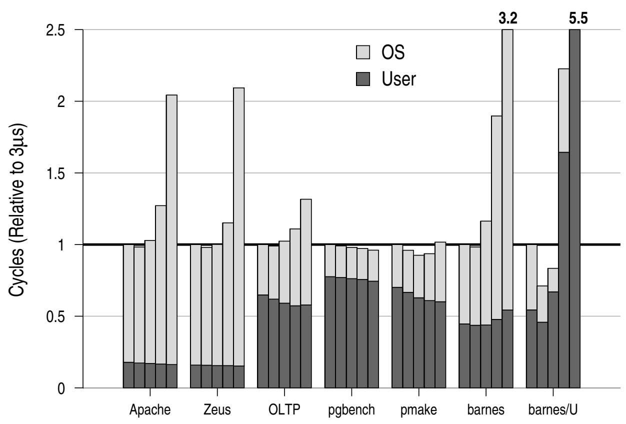

Figure 3. Normalized Runtime for Various Timeslices. *Five bars for each benchmark (from left to right) represent results for 3μs, 6μs, 17μs, 33μs and 66μs timeslices. Results are normalized to the 3μs timeslice.*

If the VCPU is currently executing, the interrupt is then delivered to the physical core. If the table reports that the VCPU is currently paused, then the interrupt is buffered until the VCPU is run again.

The final step is control logic, called the *Virtualization Controller* (VC), to oversee the mechanisms at each core. Our evaluation assumes the VC is implemented as a hardware state machine, but it would also be possible to implement it using microcode on an auxiliary core, or even one of the main cores. The VC implements the logic necessary for the applications discussed in Sections 5 and 6.

Overcommitted Virtual Machines Many examples of dynamic heterogeneity from Section 2 create a varying number of physical cores that are available, or appropriate, for a given set of VCPUs. During times when fewer cores are available than the number of VCPUs that are exposed to the OS, the cores are said to be *overcommitted*. Current software VMMs, such as VMware, do not allow the VCPUs of a *single* VM to run on cores that are overcommitted. Instead all VCPUs of a guest VM must be co-scheduled, i.e., all are run or none are. The reason is simple: the synchronization primitives in modern OSs rely on the fact that all VCPUs are executing simultaneously. This assumption can lead to livelock and/or severe performance loss when a paused VCPU is holding a kernel lock, or is the recipient of a software interrupt (CPU cross call) [38, 41].

Figure 2 demonstrates an example of this problem arising in multithreaded workloads. This figure shows the number of cores performing useful work after one of eight VCPUs was paused, and the system became overcommitted. For all workloads, the number of cores performing useful work immediately drops to seven (or lower). `Vortex`, a multiprogrammed workload with eight independent processes, remains at seven for the duration of the fault. For `artOMP`, a second core stops performing work after 2ms because it has blocked waiting on a TLB shootdown request sent to the VCPU formerly executing on the paused core. The other four workloads have much more frequent interaction among cores, causing rapid degeneration of the entire system's forward progress. For Apache and Zeus, nearly half of the VCPUs in the system stop making forward progress within 1ms.

Rapidly context switching VCPUs when overcommitted can mitigate the effects of OS synchronization. Figure 3 demonstrates the runtime of frequent VCPU context switching for a highly overcommitted system (24 VCPUs on the same 8 core system). As timeslices increase, running VCPUs spend more and more time spinning while waiting on paused VCPUs. As the figure shows, very short timeslices, such as 10μs (3 orders of magnitude shorter than a typical OS scheduling quanta), are effective at reducing these spins. Yet such frequent switching destroys cache, TLB, and branch predictor locality within each core, and even with hardware support, incurs significant overhead managing VCPU state.

Hardware Spin Detection Instead of overly frequent VCPU context switching, we propose to only perform a switch when it is necessary: when a running VCPU is spinning waiting on a VCPU that is paused. In order to avoid making any modifications to the OS, we propose a simple yet effective heuristic to identify spin loops in hardware by observing the dynamic instruction stream. The proposed heuristic relies on the observation that a program executing in a spin loop has a distinctive execution pattern: While waiting for certain events and not making any forward progress, a thread typically makes very few, if any, modifications to the program state. We can infer this lack of program state modifications from the absence of store instructions that change values in memory. Consequently, this execution pattern can be easily recognized by observing few *unique* stores committed by the program in a given interval, where the uniqueness of a store is determined by having an address or value different from other stores. To avoid false positive spin detections while searching through an array, for example, we

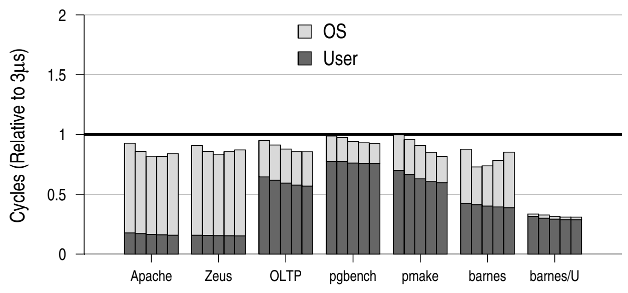

Figure 4. Normalized Runtime using the SDB. *Bars represent the same timeslices as Figure 3, and are normalized to the 3µs timeslice without the SDB.*

also check for unique load instructions (uniqueness determined by the load address only) when executing user code.

Thus, a kernel spin is detected when the number of unique stores executed within N committed instructions is less than some pre-defined threshold. On the other hand, a user spin will be detected when both unique stores and loads are less than that threshold. Sensitivity experiments demonstrate that for a period of N=1024 committed instructions, a threshold value of eight is effective to detect all known spin loops in all examined workloads with near-zero false positives.

We propose a simple hardware structure, the *Spin Detection Buffer* (SDB), to implement spin detection functionality. It employs two fully associative, eight entry content-addressable memory (CAM) structures to hold the unique stores and loads, respectively. During a given period of N instruction (N=1024 for the experiments here), each committed store (and load when in user mode) searches the appropriate CAM to determine if its address/value is unique. A unique load or store then inserts its address/value into the appropriate CAM array. Once either array becomes full, subsequent instructions need not search the CAM.

At the end of the period of committed instructions, the SDB simply checks the number of entries in each array. If there are less than eight valid entries in the store array and the VCPU is executing in the OS, the SDB indicates a spin. If there are less than eight entries in both arrays and the VCPU is executing user code, the SDB again indicates a spin. Otherwise, the arrays are flushed and it is assumed that the VCPU is making forward progress. If a user/OS mode change occurs within the period, forward progress is assumed regardless of the number of entries in the arrays.

Figure 4 shows a similar experiment as Figure 3, except that the SDB is used and the timeslice now represents a *maximum*. Should the SDB detect a spin, it will preempt and context switch VCPUs more frequently. As shown in this figure, increasing the timeslice while using the SDB does not cause the explosion in runtime that occurs without the SDB. Furthermore, increasing the timeslice improves cache locality and reduces virtualization overheads, providing runtime reductions of 10–20% for many benchmarks.

Overall, the proposed SDB technique works well for virtualized environments as it automatically detects other cases where a VCPU is not doing useful work, such as the OS idle loop and spins in user code. Together, these multicore virtualization techniques enable a multitude of applications that can adapt to, or take advantage of, dynamic heterogeneity.

5. Managing Reliability

For the past 50 years or more, software has maintained the notion that hardware is reliable. Our algorithms and software infrastructure are dependent upon reliable hardware, and the hardware we use has largely delivered on this promise. Yet maintaining hardware reliability is rapidly becoming a source of new challenges to architects and system designers, in addition to the challenges it has traditionally placed on circuit and manufacturing experts. These challenges arise as technology scales, because individual devices are increasingly susceptible to a variety of hardware faults caused by a multitude of factors, including high-energy particle strikes, manufacturing process variation, device wear-out, and temperature and voltage fluctuations [4, 5, 9, 29, 31].

These faults, and the mechanisms used to build reliable components despite them, give cores a varying ability to perform computation. Together with the rapidly changing reliability requirements of the software, these issues lead to a constantly varying pool of underlying resources. Multicore virtualization goes a long way toward increasing the effectiveness of existing reliability techniques in the presence of these changing resources, while preserving the illusion of continuous, reliable operation to the software. We now discuss how these two forms of dynamic heterogeneity arise, and how our proposed multicore virtualization can help.

5.1 Intermittent Faults

Hardware faults may manifest as *transient* faults, affecting a single transistor or wire for one cycle or less, as *permanent* faults, after irreversible wear-out damage has occurred, or as *intermittent* faults, which can occur frequently and irregularly for several cycles to several seconds or more, and then disappear for a period of time. Intermittent faults exhibit some of the worst properties of both transient and permanent faults, in that they cannot be relied upon to either go away, as required by most techniques to tolerate transient faults, or to consistently stay, as assumed by several techniques to tolerate permanent faults. Despite these challenges, several hardware schemes appear capable (with minor modifications) of detecting and recovering from a variety of intermittent faults (e.g., [11, 15, 19, 34]).

We argue that suspending the use of a core when one of these mechanisms detects that a burst of faults is beginning (or is expected to occur) can improve the overall reliability of the system by reducing the opportunity for one or more faults within a burst to go undetected. Yet naively suspending a single core while it is executing software, even for

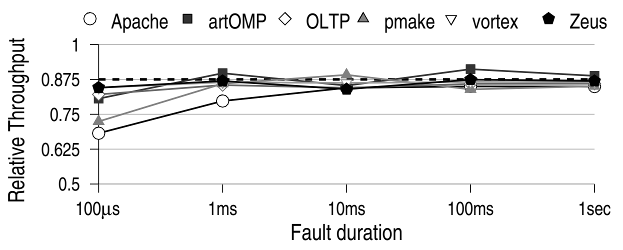

Figure 5. Throughput of Overcommitting During a Fault

a few milliseconds, can have significant performance consequences. Figure 2 illustrates one example of this consequence: cascading livelock as the software running on other cores attempts to synchronize with the paused core.

Traditionally, system software is responsible for determining which cores are actively running software threads. Instead of simply pausing execution, another viable option to suspend the use of a core is to interrupt the OS (or hypervisor), and ask it to reconfigure itself to only use the remaining fault-free cores. Some current OSs (such as Solaris) and hypervisors (such as those that run on the IBM zSeries) already contain this functionality [2, 37]. However, software reconfiguration can take several milliseconds, and can cause high overheads for frequent intermittent faults of short duration [42].

Multicore virtualization offers a way to quickly adapt to a varying number of usable cores. By using hardware-based checkpoints (e.g., [36]), the state of the VCPU running on a temporarily unusable core can be recovered and migrated to a different core. Since two VCPUs are likely to be sharing one of the remaining cores, the system becomes overcommitted. Figure 5 shows the throughput of the overcommitted system *during* faults of various durations, compared to a fault free machine. This technique does incur some overhead for the shortest duration faults, due to checkpoint recovery, VCPU migration, and cold cache misses. But that overhead is small and is quickly amortized for longer fault durations. Overall, the throughput is within a few percent of the expected performance of a seven core system, despite the fact that the OS still believes it is running on eight cores. In addition to good throughput results, multicore virtualization allows us to adapt to the effects of intermittent faults, while maintaining fairness among VCPUs, preserving low software transaction latency, incurring essentially zero fault-free cost, and gracefully handling multiple concurrent failures [42].

5.2 Mixed-Mode Reliability

Suspending cores to tolerate intermittent faults is one cause of changing resource configurations. We observe that the reliability requirements of code can change dynamically as well, leading to another cause of varying configurations. For example, certain applications and users already desire high reliability and the peace of mind that comes with the use of Dual-Modular Redundancy (DMR), where two cores are (loosely) joined together to redundantly execute a software thread from one VCPU [1, 3, 24, 33]. Trends in hardware reliability are likely to encourage more and more users to seek such levels of reliability in future generations of processors. Yet the reliability of DMR comes with significant penalties (2–4X) in terms of per-thread performance, throughput and power efficiency, and many applications which are less sensitive to moderate levels of hardware faults are unwilling to pay this price. Running both types of software on the same machine at the same time results in a system where the number of cores required to execute a single VCPU changes dynamically depending on what software the OS schedules onto that VCPU.

To support such changing requirements, we propose a *Mixed-Mode Multicore* (MMM) system, where certain applications (or portions of applications) run in high performance mode using a single core, while other applications (including the system software) run in a highly reliable mode using DMR [43]. Media applications, for example, tend to be insensitive to moderate levels of hardware faults [30], but a user may be willing to sacrifice a certain degree of performance to ensure the integrity of their financial data. An MMM can allow a desktop user to run both types of applications at the same time.

Conceptually, an MMM is simple: use DMR for software that needs it, and turn off DMR for software that prefers to execute in performance mode. The required software interface is a single register per VCPU specifying whether reliability is needed or not. When the privileged software is about to context switch to an application which requires high performance, it writes this register to indicate the requirements of the software running on that VCPU.

In practice, an MMM introduces a number of challenges, but one in particular exposes the need for multicore virtualization: the desire to use both cores of a redundant pair to execute independent VCPUs when running in high performance mode. Independent use of these cores is necessary to improve throughput during performance mode. The problem is that VCPUs dynamically and independently switch modes depending on what software the OS has scheduled onto that core.

Multicore virtualization can address this issue by allowing the chip to flexibly assign VCPUs to cores. The chip exposes as many VCPUs as there are cores, and then operates in an overcommitted manner when one or more VCPUs enter reliable mode. An overcommitted mixed-mode system is depicted in Figure 6. Here, one VCPU (V2) is executing a software threads that requires reliability, and is executing redundantly on cores C2 and C3. V3 is paused since there are no cores available to execute it. The other VCPUs are all executing threads that require performance.

Figure 7 demonstrates the ability of a mixed-mode multicore server to provide differentiated service to different ap-

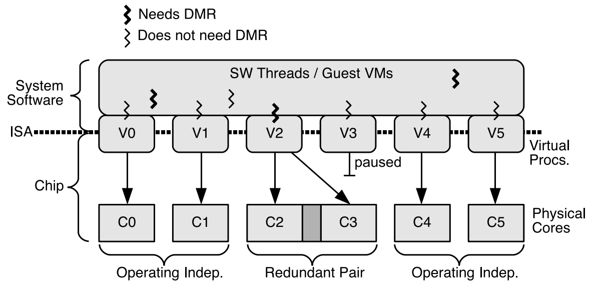

Figure 6. Improving Throughput in a Mixed-Mode Multicore by Overcommitting Cores

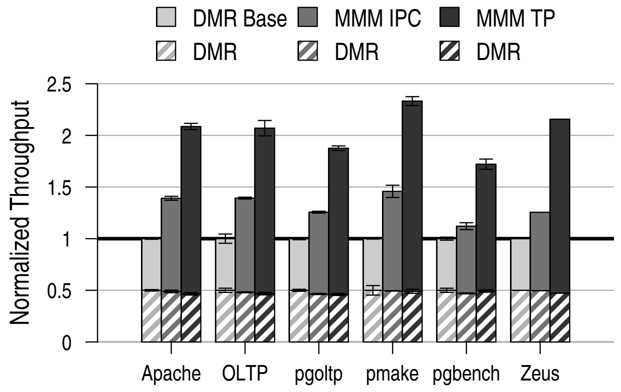

Figure 7. Throughput of Mixed-Mode Execution

plications. In this experiment, we are modeling an MMM running consolidated server workloads, where one guest virtual machine (VM) requires reliability, and a second guest VM requires performance mode. The striped bars at the bottom represent the normalized throughput of the guest VM that requires the high reliability of DMR. The solid, top bars represent the guest VM that does not require such high reliability. Unlike the other experiments in this paper, our target multicore has 16 cores.

In a traditional consolidated server, if one guest VM required reliability, then all guests would need to run with DMR to protect the integrity of the reliable VM and the VMM itself. The left set of bars (labeled *DMR Base*) thus represents the baseline, where reliable, DMR mode is used for both VMs. The second set of bars, labeled *MMM-IPC*, represents a MMM where multicore virtualization is not used and the unused redundant cores are allowed to idle. Due to the IPC overhead of DMR execution, the high-performance guest VM observes 25–85% speedup over the full DMR configuration. The third set of bars, labeled *MMM-TP*, represents the an MMM system designed to improve throughput by using multicore virtualization to better utilize all available cores to execute additional VCPUs during performance mode. For scalable applications, such as these commercial workloads, improvements in throughput can be significant using MMM-TP, where the first VM now independently executes twice as many VCPUs. This high-performance VM observes speedups of 2.4–3.6 due to the combined effect of per-VCPU IPC increase, and additional throughput from more VCPUs. Speedups of this VM over the static MMM configuration are 1.8–1.9. The throughput of the machine overall increases by 1.7-2.3X.

6. Enhancing Locality

The two examples of the previous section address the challenges of adapting to dynamic heterogeneity created either by the varying capabilities or configurations of the cores. In this section, we discuss two proposals which actively *create* dynamic heterogeneity, and then use it to an advantage.

6.1 Computation Spreading

In the traditional mode of assigning computation from multiple threads to multiple processors, an entire software thread — including any operating system calls it makes — is assigned to a single processor for execution. This was, perhaps, the only practical approach for traditional multiprocessors built from multiple chips. But since there is commonality among the computation performed by the different threads, this distribution leads to inefficient use of the microarchitectural structures of the individual processing cores, such as private instruction caches and branch predictors. With support for multicore virtualization, ample opportunities exist for alternate solutions, without requiring changes to software. We propose *Computation Spreading (CSP)* as a method for distributing different fragments of a thread's computation across multiple processing cores [7]. We define a *computation fragment* as an arbitrary portion of a dynamic instruction stream. Conceptually, CSP aims to collocate similar computation fragments from different threads on the same core while distributing the dissimilar computation fragments from the same thread across multiple cores. Each core thus becomes dynamically and temporally specialized for executing a set of specific computation fragments by retaining the states (such as instruction cache contents and branch predictor entries) necessary to perform each computation efficiently.

After examining the code reuse characteristics of four multithreaded server workloads, we find that most instruction blocks are accessed by many, if not all, cores on a chip. This fact implies that they all execute similar computation fragments (albeit at different times), and the canonical model of work distribution leads to inefficient use of the aggregate cache space.

As a specific application of CSP, targeting server workloads, we propose two assignment policies which separate the execution of system calls and interrupt handlers from the execution of user code, and distribute these two dissimilar computation fragments to different cores. *Thread Assignment Policy* (TAP) prefers to run the OS (or user) portion of a thread on the same core repeatedly, aiming to reduce OS

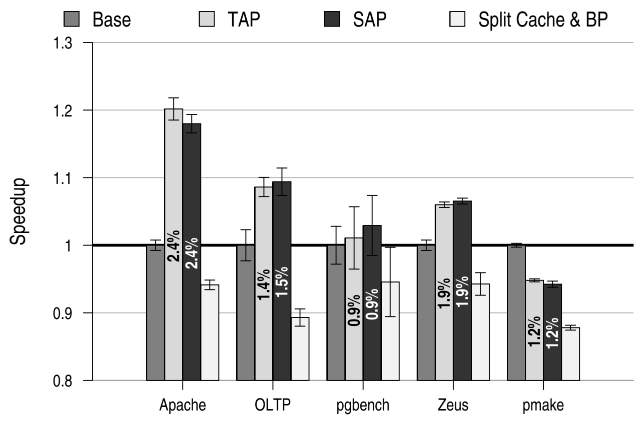

Figure 8. Performance Comparison. *Labels on TAP and SAP bars represent the overhead of multicore virtualization.*

and user interference while maintaining data and instruction locality for each software thread; *Syscall Assignment Policy* (SAP) prefers to run a particular system call (e.g., *read()*) on the same core repeatedly, regardless of which thread made the call, aiming to further improve instruction locality and take advantage of any data structures shared among multiple dynamic instances of the same system call. Both provision a subset of the processing cores for executing user code, and the remainder for the OS.

Unlike previous research on separating OS and user execution, which primarily considered one or more single-core processors [22], TAP and SAP are able to alleviate the interference of separating dissimilar tasks *and* benefit from the symbiosis of collocating similar tasks. The two specific assignment policies (TAP and SAP) we propose both spread user and OS execution across different cores. For four workloads, TAP and SAP reduce L2 instruction misses by 27–58%, L2 load misses by 0–19%, and branch mispredictions by 9–25%, resulting in a performance improvement of 1–20%, as seen in Figure 8. The results for `pmake`, a fifth, non-server benchmark, are not as favorable.

As a comparison, the fourth bar of Figure 8 shows a configuration where we separate each private cache and branch predictor into separate structures used by the OS and user code, while keeping the aggregate sizes the same. As evident from this figure, simply separation to eliminate interference leads to worse performance.

While separating user and OS execution is interesting for OS-intensive workloads, Computation Spreading has much potential for further improvement for these and other classes of workloads by more intelligently spreading independent computation within user or OS execution. We leave an examination of these ideas for future work.

6.2 Dynamic Core Partitioning

Server consolidation refers to the process of moving two or more services from multiple, separate machines onto one physical machine [2, 40]. A *Virtual Machine Monitor* (VMM) is responsible for sharing the physical resources of the consolidated server, including the processing cores, among multiple VMs. While economically sound, this act can create significant interference in the per-core predictive structures. In the same vein as Computation Spreading, consolidated servers can also benefit from preserving locality by mapping similar types of computation onto the same cores.

Conflicting Objectives Providing the performance expectations are met, maximizing efficiency is the major goal of a consolidated server. It requires the VMM to adapt to varying demand on the services of each guest VM, but also maintain the locality of per-core predictive structures, such as caches, TLBs, and branch predictors — in effect, creating dynamic heterogeneity by specializing these predictive structures for a particular type of task. However, adapting to demand and maximizing locality are at odds due to the need of most VMMs to *gang schedule* the VCPUs belonging to each VM. Gang scheduling, or co-scheduling [26], simply refers to the policy of either concurrently running *all* VCPUs of a given VM, or none of them. Gang scheduling is used by VMware ESX server [39] and Cellular Disco [12], among others, in order to avoid the serious synchronization issues that arise when not all of the VCPUs of a given guest VM are concurrently executing (see Section 4).

In order to adapt to changes in demand, a consolidated server can *time partition* the cores of the machine among guest VMs, and adjust the timeslices of each VM according to demand. The problem with gang scheduling, however, is that by timesharing each core among unrelated operations, cache, TLB, and branch predictor locality is destroyed. As an alternative, statically partitioning the cores between multiple VMs achieves the desired locality, and dynamic heterogeneity, but prevents a single VM from using all of the cores of the chip during periods of high demand.

Overcommitting and Dynamically Partitioning To resolve the conflicting objectives that plague gang scheduling, we propose to use *dynamic partitioning* with multicore virtualization to enable the ability to respond to changes in both workload demand and the changing characteristics of the underlying chip, as well as the ability to optimize performance, efficiency, and isolation, and quickly adapt to other dynamically changing characteristics of the chip. To do this, a number of VCPUs equal to the number of cores is exposed to each guest VM, allowing it to execute those VCPUs on the entire machine when necessary during peaks in demand. Then, during periods of normal demand, the physical cores are dynamically partitioned among guest VMs, allowing the VMM to only execute a subset of each VM's VCPUs at a time. Since guest VMs are allocated fewer cores than VCPUs, the guest VMs becomes overcommitted.

Results Figure 9 shows the overall performance of each VM for the consolidated workloads for the four scheduling

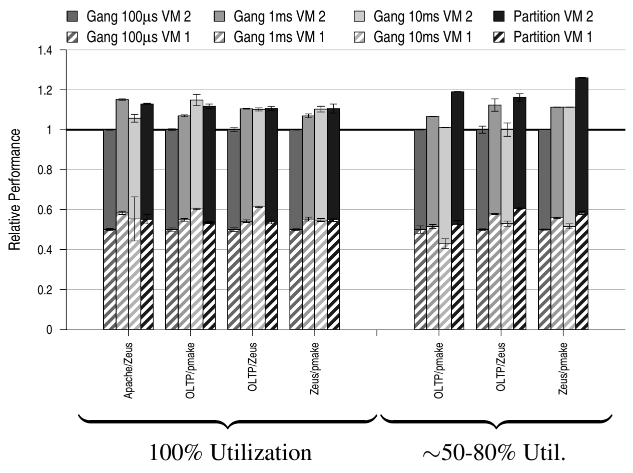

Figure 9. Performance of Different Scheduling Policies. *Results are normalized to gang scheduling with a 100μs timeslice (higher is better).*

policies. Total performance is normalized to gang scheduling at 100μs. Speedup is simply the average speedup of the two VMs, though the graph breaks down the speedup component from each VM. Striped bars represent VM 1, and solid bars, VM 2. Error bars represent the 95% confidence interval, which is calculated independently for each VM.

As expected from improving cache, TLB and branch prediction locality, overcommitting and dynamic partitioning (the rightmost bar) provides speedups from 10–20% for all but one full utilization workloads, and a slightly higher 18–25% for the lower utilization workloads, where it can recover some of the time VCPUs spend idle. Using a 1ms timeslice for gang scheduling (second bar) improves performance in all experiments by 5–19%, although this comes at the price of additional latency on each request. In many cases, gang scheduling with a 10ms timeslice (third bar) does not continue to improve performance compared to a 1ms timeslice.

Although the speedups for dynamic partitioning are modest, these experiments demonstrates that more flexible scheduling algorithms can be important by providing the throughput of a long gang scheduling timeslice, while delivering the expected transaction latency of a short timeslice.

6.3 Hardware or Software Support?

Both Computation Spreading and dynamic partitioning of consolidated servers can be performed with an unmodified, traditional software VMM, by implementing the multicore virtualization underneath the ISA. But unlike the reliability examples in Section 5, both of these ideas are alternately very amenable to software support similar to, for example, Cohort Scheduling [20], SEDA [44], and STEPS [16]. All of these alternate projects use additional software complexity to create and exploit dynamic heterogeneity.

Similarly, using a para-virtual VMM environment, the hardware/software interface can be changed to support virtualization. With para-virtualization, a VMM could be created to allow the cores used by a single guest VM to be overcommitted relative to that VM's VCPUs, and hence gain the benefits of the proposed dynamic partitioning.

7. Conclusions

The continued exponential growth in the number of transistors on a chip presents several challenges to computer architects. In order to simplify the problem for hardware designers, most computer manufacturers have switched to building multicore processors. However, multicores greatly complicate software's traditional task of assigning computation to cores.

Of particular concern, this paper identifies *dynamic heterogeneity* as a growing trend for multicore designs. Dynamic heterogeneity can occur, even among physically homogeneous cores, from reliability, power, or thermal conditions, different cache and TLB contents, or even changing resource configurations. This heterogeneity creates a rapidly varying pool of resources, resulting in uncertainty about which cores are available or most appropriate for running a given computation at a particular time. Current multicore processors, and many proposed designs for future multicore systems, simply pass this uncertainty up to the software. Yet software's need for extracting concurrency in the first place is a big enough challenge in the multicore era. Continuing to require software to explicitly manage the use of all cores in order to express that concurrency to the hardware is an additional burden which is both undesirable and unattainable.

In this paper, we argue for hardware taking a more active role in the management of its own resources, enabled by our proposed hardware techniques to virtualize the cores of a multicore processor. Multicore virtualization allows hardware to transparently remap the *virtual processors* (VCPUs) exposed even to a single operating system (OS) to any subset of physical cores. We demonstrate that by using these techniques, a processor can manage the changing resource configurations created by hardware faults and software's requirements, and can improve locality of per-core predictive structures through flexible assignment of computation to cores. We further believe that multicore virtualization can be useful for many other applications as well.

By decoupling decoupling the tasks of low-level core management and high-level concurrency extraction, we believe multicore virtualization will have a substantial impact on system design, facilitating the evolution of future "many-core" systems.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) grants CCF-0702313 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin.

References

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proc. of 34th ISCA*, 2007.

[2] W. Armstrong, R. Arndt, D. Boutcher, R. Kovacs, D. Larson, K. Lucke, N. Nayar, and R. Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM J. Res. & Dev.*, 49(4/5), 2005.

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *Proc. of 2005 DSN*, 2005.

[4] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. of 40th DAC*, 2003.

[5] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *J. of Solid-State Circuits*, 37(2):183–190, Feb 2002.

[6] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of 7th HPCA*, 2001.

[7] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proc. of 12th ASPLOS*, 2006.

[8] K. Chakraborty, P. M. Wells, and G. S. Sohi. A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs. Technical Report CS-TR-2007-1607, University of Wisconsin-Madison, Aug 2007.

[9] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[10] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. pages 102–103, Feb. 2007.

[11] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of 36th MICRO*, 2003.

[12] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proc. of 16th SOSP*, 1999.

[13] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor. In *Proc. of 17th Hot Chips*, 2005.

[14] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Tech. J.*, Q1, 2001.

[15] S. N. Hamilton and A. Orailoglu. Transient and intermittent fault recovery without rollback. In *Proc. of 13th Defect and Fault-Tolerance in VLSI Sys.*, 1998.

[16] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proc. of 30th VLDB*, 2004.

[17] E. Ípek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proc. of 34th ISCA*, 2007.

[18] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of 31st ISCA*, 2004.

[19] C. LaFrieda, E. Ípek, J. F. Martínez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of 2007 DSN*, 2007.

[20] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track USENIX Annual Technical Conference*, 2002.

[21] J. Laudon. Performance/watt: the new server focus. *Comp. Arch. News*, 33(4):5–13, 2005.

[22] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and improving operating system effects in control flow prediction. In *Proc. of 10th ASPLOS*, 2002.

[23] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Comp.*, 35(2):50–58, Feb 2002.

[24] D. McEvoy. The architecture of tandem's nonstop system. In *Proc. of ACM 1981 Conf.*, 1981.

[25] D. Nellans, R. Balasubramonian, and E. Brunvand. A case for increased operating system support in chip multi-processors. In *Proc. of 2nd IBM Watson P=ac^2*, 2005.

[26] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Distributed Computing Systems*, 1982.

[27] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proc. of 11th ASPLOS*, 2004.

[28] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proc. of 30th ISCA*, 2003.

[29] Semiconductor Industry Association. International technology roadmap for semiconductors: Executive summary, 2005.

[30] J. W. Sheaffer, D. P. Luebke, and K. Skadron. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In *Proc. of 21st Eurographics GH*, 2006.

[31] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of 2002 DSN*, 2002.

[32] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proc. of 30th ISCA*, 2003.

[33] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[34] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of 39th MICRO*, 2006.

[35] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of 22nd ISCA*, pages 414–425, 1995.

[36] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of 29th ISCA*, 2002.

[37] Sun Microsystems, Inc. Sun fire high-end and midrange systems dynamic reconfiguration user's guide. Viewed 12/19/2007.

[38] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of 3rd Virt. Mach. Research and Tech. Symp.*, 2004.

[39] VMware. ESX Server - best practices using VMware virtual SMP. Viewed 5/03/2006.

[40] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. of 5th Symposium on OSDI*, 2002.

[41] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. of 15th PACT*, 2006.

[42] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. In *Proc. of 13th ASPLOS*, pages 255–264, 2008.

[43] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. In *Proc. of 14th ASPLOS*, 2009.

[44] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[45] K. Wonyoung, G. Meeta, W. Gu-Yeon, and B. David. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proc. of 14th HPCA*, February 2008.