

Design of a Custom VEE Core in a Chip Multiprocessor

Dan Upton
Master's Project

Abstract

Chip multiprocessors provide an opportunity for continuing performance growth in the face of limited single-thread parallelism. Although the best design path for such multiprocessors remains open, application-specific core designs have shown promise. This work considers the design of such an application-specific core for a virtual execution environment. We use Pin, a widely-used dynamic binary instrumentation framework, as a representative process-level VEE. Through a combination of microarchitectural simulation and hardware performance counters, we profile the VEE in terms of cache behavior, functional unit usage, and branch predictor behavior, and compare its performance over several inputs to the performance of benchmark applications. We discuss trade-offs in layout at the core level as well as support structures for and opportunities created by moving the VEE's execution to a separate core. We show running the VEE on our specialized core uses up to 15% less power per cycle and up to 5% less power overall than running the same VEE on a general-purpose core.

1. Introduction

Increasing the clock frequency is no longer a guaranteed way to increase the performance of modern processors. Wire delay, fabrication limitations, power consumption, and heat dissipation all decrease the effectiveness of increasing transistor count and frequency. Furthermore, techniques for extracting greater levels of ILP from single-threaded applications have shown diminishing returns as penalties for mispredictions and other pipeline restarts increase.

Thus, recent processor designs have looked to take advantage of thread- or task-level parallelism, such as simultaneous multi-threading processors [38] and chip multiprocessors (CMPs) [34]. Such designs can take advantage of higher transistor counts by replicating control structures and functional units, allowing multiple threads of execution to run simultaneously and increase overall throughput.

The best design path for chip multiprocessors remains an open question. A straightforward solution, taken by many current multicore architectures [19, 23], is to have several identical cores, save for flaws in the fabrication process [17]. However, other results have suggested heterogeneous designs can improve performance, either in terms of power consumption [24] or throughput [27]. Furthermore, research has shown promising results in increasing per-

formance by specializing cores of such a heterogeneous processor, designing different cores for different classes of applications [26].

In this work, we consider the design of a core specialized for execution of virtual execution environments (VEEs). Virtualization has seen a resurgence in recent years, with applications ranging from security [16, 22] and instrumentation [30] to optimization [5, 10] and JVMs and binary translation for ISA portability [1]. These systems frequently add some execution overhead, which may be mitigated by moving the execution off-core. Furthermore, they may vary enough from other general classes of applications that a specialized core could save on die area, power consumption, and heat dissipation, as compared to a general-purpose core.

To guide our design, we profile Pin [30], a widely-used dynamic binary instrumentation framework, as a representative VEE. We first read hardware performance counters via PAPI [9] to characterize our VEE in terms of floating point hardware usage, branch predictor behavior, and miss rates in the various caches, and compare these characteristics to various benchmarks from the SPECINT2000 suite. From this, we show our VEE has a very small percentage of floating-point instructions, has a higher percentage of branches but a lower misprediction rate, and a lower cache miss rate, than the average of the benchmarks.

We then consider the effect of changing microarchitectural structure sizes through simulation on SimpleScalar [4]. Through our simulations, we show we can decrease the complexity of several of the structures without significantly impacting performance. We then use Wattach [8] to calculate the power usage of a core designed based on our characterizations, showing up to a 15% improvement per-cycle and up to a 5% improvement in overall energy consumption.

Finally, we consider the design space for a multicore chip including one or more private or shared VEE cores, as well as support at the hardware level for running the VEE on a separate core. Beyond this, we discuss the possibilities opened by moving the VEE execution to a separate core. For instance, separating the execution mitigates performance overhead due to the VEE and application sharing the same hardware context. It also allows parallelizing VEE tasks, such as JIT compilation, with application execution.

The rest of the paper is organized as follows. Section 2 provides background material related to chip multiprocessors and virtual execution environments, including an overview of Pin. Section 3 presents and discusses several characterizations, leading to a final core design, and demonstrates power savings of our design compared to current processors. Section 4 goes on to discuss floor planning options and trade-offs, extensions to support executing a VEE on a separate core, and opportunities opened by this separation. We then present related work in Section 5, and conclude and discuss future work in Section 6.

2. Background

This section provides a brief background of chip multiprocessors and virtual execution environments. After an overview of VEEs in

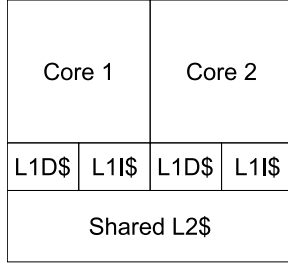


Figure 1. A straightforward CMP design, similar to the Intel Core Duo, with two separate cores with separate level-1 caches and a shared level-2 cache.

general, we provide more detail on the virtual execution environment we use in our studies.

2.1 Chip Multiprocessors

Olokotun *et al.* [34] were among the first to advance the idea of single-chip multiprocessors. CMPs can take better advantage of thread- and task-level parallelism than simultaneous multithreading processors with less physical complexity, because no additional logic is required to distinguish instructions from multiple threads or to ensure fair scheduling. CMPs can also provide better performance on parallel applications, due to the on-chip networks being much faster than moving between chips or even physical machines.

Many current CMPs, such as the Intel Core Duo [19] (illustrated in Figure 1) and Sun Niagara [23], take the design approach of replicating similar cores on a chip, frequently with some level of shared cache. However, these cores may be different, either due to intentional design decisions or due to errors in the fabrication process. Such CMPs are called heterogeneous or asymmetric CMPs.

Balakrishnan *et al.* [6] demonstrated that heterogeneous CMPs can produce sub-optimal behavior if neither the application nor the operating system scheduler are aware of such heterogeneity. However, Kumar *et al.* [24, 27] demonstrated the potential for improved throughput and decreased power consumption via heterogeneous CMP design. In both of these cases, all cores were essentially general-purpose processors, varying primarily in either clock rate or in processor family generation.

Further work by Kumar *et al.* [26] considered heterogeneous design from the ground up—rather than selecting from different generations of the same family of processors, different cores were designed based on the characteristics of different classes of applications. This design philosophy, which we follow in the work presented here, was shown to potentially improve performance over that which could be obtained developing a heterogeneous CMP only from general-purpose processors.

2.2 Virtual Execution Environments

Virtual execution environments can be grouped into two general categories, as illustrated in Figure 2: system VEEs, where the VEE runs below the OS and essentially virtualizes all applications as well as the OS, and process VEEs, where just a single application runs on each instance of the VEE. The work presented in this paper will focus primarily on process VEEs. However, system VEEs perform some similar tasks, so the work presented here can be used as a starting point for core design for system VEEs. Thus, we will briefly discuss system VEEs as pertains to our work.

System virtual execution environments provide virtualization of the hardware to the operating system and all applications. They may either reside between the hardware and the base operating system, as in Xen [7] and Transmeta’s Code Morphing Software [12] and depicted in Figure 2(b), or may run as an application in an-

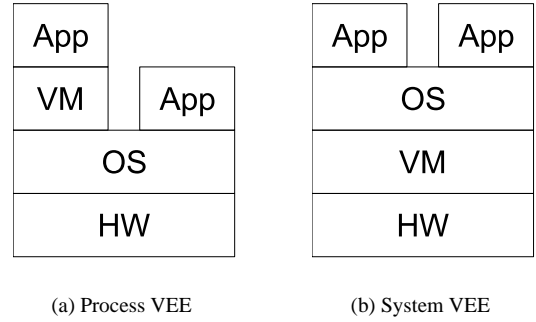


Figure 2. Abstracted views of process and system virtual execution environments. In the process VEE case, the VEE runs above the OS and a single application runs on each instance of the VEE, while in the system VEE case, the VEE runs below the OS and all applications.

other operating system, such as VMWare [37]. System VEEs have applications ranging from security [13] to debugging support [21] to architectural simulation [40]. Some hardware support already exists for system VEEs, such as Intel’s VT [18] and AMD’s SVM [2], which provide support at the ISA level.

Process virtual execution environments generally run as a layer between the operating system and an application, with only one application per instance of the VEE. This allows for more fine-grained control of what is executed under control of the virtual execution environment, and what in particular the VEE does with that application. For instance, one application may run under an instance of a VEE configured to dynamically instrument the guest application [30], while a different instance may enforce a security policy on the guest application [22]. These systems generally run in the same address space and hardware context as the guest application, which introduces overhead; attempting to reduce this overhead by running the VEE and the application in separate hardware contexts is a major motivation of this work. Examples of process VEEs include Dynamo [5], DynamoRIO [10], Strata [36], Valgrind [32], and Pin [30], which we use in this work. Many of the aforementioned systems share implementation details with Pin, which we describe in the next section.

2.3 Pin

Pin begins by injecting itself into the application. Once it has control, it uses a just-in-time compiler (JIT) to recompile code into traces, or superblocks, speculatively generated by following code until reaching an upper limit on instructions or an unconditional jump. This code can be modified to insert user-defined instrumentation, and then is cached so that further executions of the same code will not require intervention by the VEE. Traces generally end with a jump; if the target of a trace has also been cached, the two can frequently be linked—except for indirect branches—to allow execution of multiple traces without intervention by the VEE. Other systems may only cache basic blocks instead of superblocks, or may start with basic blocks and then build traces later based on basic block execution frequency.

When a branch target does not exist in the code cache, control must be returned to the VEE to let it JIT the next section of code. Control may also return to the VEE in the event of an indirect branch, if the various methods used by different systems fail to find the correct target. This return of control to the VEE, called a context switch, requires saving away the entire architectural state of the application and then restoring it when resuming execution of

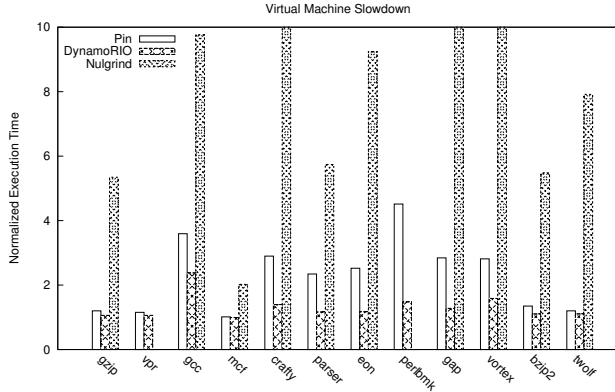


Figure 3. Overheads of running the SPEC2000 integer benchmarks under control of several virtual execution environments. Bars represent execution time normalized to native.

the application. This is different from an OS-level context switch, since the VEE an application are running in the same address space; it refers to the need to switch from the application’s register set to the VEE’s registers and back.

On many applications, the overhead of compiling and transitioning between the application and VEE can largely be amortized over the full run of the application. When the application reaches a steady state and the working set of code has been fully cached, the application can run at roughly native speed, modulo any instrumentation code. However, short-running applications, applications with little code reuse, and applications with large numbers of indirect branches require more frequently interaction with the VEE and thus have a larger overhead. As an example, Figure 3 shows the overhead of running the SPEC2000 integer benchmarks under control of Pin version 4229, DynamoRIO [10] version 0.9.4, and Valgrind [32] version 3.2, on a hyperthreaded 3.2GHz Xeon with 2GB of RAM running CentOS Linux 4.4. The Valgrind results represent the Nulgrind tool, which does not modify the application code. The runtimes reported are normalized to native execution. Results for `vpr` and `perlbnk` in Nulgrind are omitted due to compare errors. DynamoRIO’s overhead is lower because it performs some optimizations, and Nulgrind’s overhead is higher because it does not link in its cache, requiring a context switch at the end of every trace. In general, reducing this overhead via hardware support is a major motivation of the work presented in this paper. However, this paper only focuses on the hardware design; engineering a VEE to take advantage of the hardware to see the overhead reduction is future work.

3. VEE Characterization

In this section, we will present results and commentary on a characterization of virtual execution environment behavior. We begin with an explanation of the experimental setup, then follow with characterizations of VEE performance in terms of L1 and L2 cache performance, floating-point and branch prediction hardware usage, and branch misprediction rate. These structures were chosen as architectural structures that are easy to modify and have use patterns that vary by application. We then use the results of these experiments to guide our design, which we will also show yields power savings compared to existing systems.

```

section .data
hello db "Hello World",0xa

section .text

global _start
_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, hello
    mov edx, 12
    int 0x80
    mov eax, 1
    mov ebx, 0
    int 0x80
    hlt

```

Figure 4. 8-instruction kernel for testing Pin on SimpleScalar-x86, by invoking `sim-outorder-x86 pin -- hello.s`.

Table 1. Baseline SimpleScalar-x86 configuration.

Parameter	Value
Processor width	8
Fetch queue size	16
Branch predictor	Combined predictor with 16K-entry meta-table, 2-leve predictor with 16K entry L1, 16K entry L2, 14-bit history XORed with address
BTB size	512 sets, 4-way
RAS size	8
RUU size	128
L1 caches	64K, 4-way, 32B blocks
Unified L2 cache	512K, 4-way, 64B blocks
Functional units	6 int ALU, 2 int mult, 4 FP ALU, 2 FP mult

3.1 Experimental Environment

All of the results in this paper were collected using versions of Pin built from source kit 4229 with optimization level 3. We use a combination of hardware performance counters and simulation on SimpleScalar [4]; this allows us to get data representative of real hardware with no performance penalty and also to observe the impact of varying microarchitectural details.

Results based on performance counters were collected using PAPI [9] v3.2.1 and the `perfctr` Linux kernel patch. Two different machines were used for collecting performance counter numbers, due to restrictions on which counters were available on each processor. Data for floating-point hardware usage, branch prediction hardware usage, branch mispredictions, L1 instruction cache performance, and L2 cache performance were all collected on a 3.2GHz Xeon with hyperthreading enabled. The Xeon does not have a direct way to collect L1 data cache accesses, only data cache misses; to collect that data, we used a 733MHz Pentium III. Thus, the miss rates of the L1 data cache should be considered only in their own context and not with respect to the other data.

For experiments where we varied hardware structure sizes, we used SimpleScalar-x86 [4] to perform cycle-accurate simulations. Table 1 outlines the baseline simulated hardware.

One issue with collecting the performance numbers for the VEE is separating the performance characteristics of the VEE from the characteristics of the application. Separating the performance characteristics is necessary to ensure only the behavior of the VEE is used to influence the core design. It is difficult to draw conclusions from data collected at a coarse level (*e.g.*, performance num-

bers collected only at the granularity of an entire execution) for two main reasons. First, the VEE behaves differently depending on the application it is executing. Second, the VEE can affect the microarchitectural characteristics of the application, such as causing conflicts in the cache or branch history.

For many of the results collected on SimpleScalar, we address this issue by executing the short application in Figure 4 under control of the VEE. This ensures that the vast majority of instructions executed on SimpleScalar are from the VEE, and thus the performance data is indicative of the VEE’s execution. However, executing such a short application means the performance data is primarily representative of startup and shutdown of the VEE with relatively little JIT compilation and code cache management activity. In the following sections, we will discuss the causes of differences between results collected from SimpleScalar and those collected from the hardware performance counters.

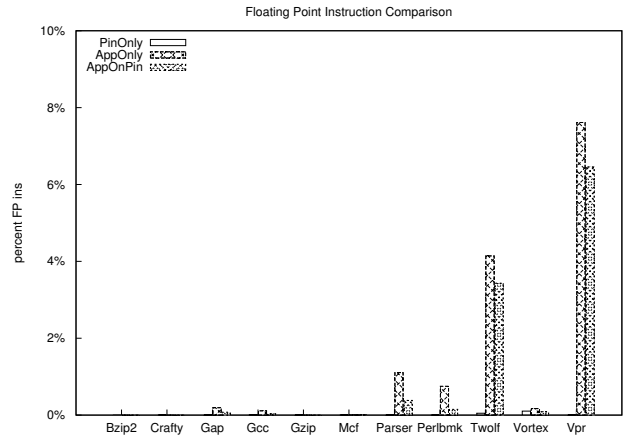
For results collected from performance counters, we modified Pin to start the counters in its `main()` function and stop them prior to shutting down, and also to start the counters when entering the VEE and stop them when entering the code cache. This captures all of the behavior of the VEE except for the initial injection. Using performance counters has the benefit that results reflect actual hardware. However, the PAPI code used to start and stop the performance counters affects the reported results. In the case of coarse-grained measurement—starting the counters at the beginning of execution and stopping them at the end—these additional instructions (approximately 1500) and memory references are essentially noise. When turning them on and off more frequently, this could add a larger overhead; however, even in the most extreme case of executing `gcc` under Pin’s control, requiring starting and stopping the counters roughly 37000 times and costing 1500 instructions per start/stop pair, this is still less than half of a percent of the total number of instructions executed by the VEE. Thus, even the finer-grained performance counter switching still provides results indicative of the VEE itself and not of the application it is executing or of the performance-monitoring tool.

In the following sections, the performance of the VEE will be shown for all of the SPEC2000 integer benchmarks and compared to the native performance of each of those applications. This serves two purposes. First, since VEEs run a wide variety of applications, it gives us a representative set of applications to measure the VEE’s varying performance. Secondly, it can highlight where the VEE is similar to other applications. For instance, as will be discussed below, Pin’s L2 cache miss rate shows a dependence on input, similar to `perl1bmk`. Many of the graphs also show performance data for the VEE and application together as another point of reference. This data can give a rough estimate of the impact of aliasing in hardware structures and how running under control of the VEE can affect the application’s performance.

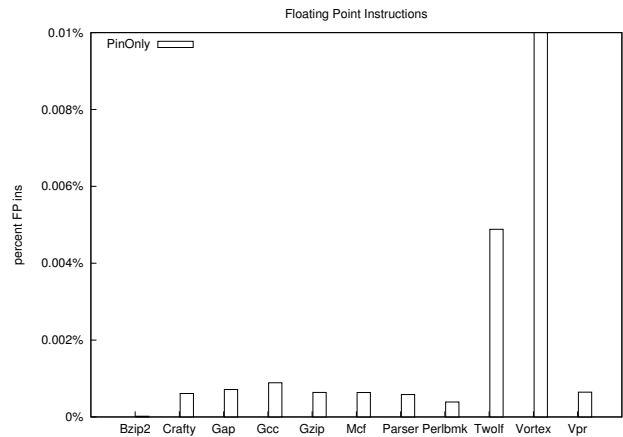
3.2 Floating-Point Hardware Usage

The first feature we investigated was the VEE’s floating-point hardware usage. Figure 5 shows a comparison of the percentage of floating-point instructions executed by Pin alone, the application alone, and the application running under Pin’s control, as reported by the performance counters.

Figure 5(a) shows that in general, the percentage of floating-point instructions executed in the VEE is greatly dwarfed by the percentage of floating-point instructions executed by some of the other benchmarks; Figure 5(b) zooms in and shows only the percentages for Pin. The value for `vortex` is only slightly higher than the graph’s range of 0.1%. Evaluating why `vortex` and `twolf` are outliers would be an interesting further study; specifically, whether the additional code they exercise is frequently exercised by other (non-benchmark) applications. In general, however, these results



(a) Pin, application, and application on Pin



(b) Pin only

Figure 5. Comparison of percentage of floating-point instructions executed by Pin managing the application, the application natively, and the application and Pin together, collected from performance counters.

suggest floating-point hardware is not a determining factor of VEE performance.

There are multiple options for providing floating-point ability to the VEE core when necessary. Naturally, the floating-point pipeline could be left on the core, despite its low utilization. Alternately, floating-point instructions could be emulated. Finally, a conjoined core approach, as suggested by Kumar *et al.* [25] could be taken. If each general-purpose core had a separate VEE core, as will be discussed later, the VEE core could share the floating-point pipeline. Their work suggested this only caused severe performance degradation if both cores were running floating-point intensive applications. Here we would have at least one application, the VEE, that was floating-point light. Furthermore, if the VEE is not parallelized with the application, the application would be stalled and thus would not conflict with the VEE using the floating-point hardware. Thus, a VEE core can be constructed without a dedicated floating-point pipeline without severely impacting performance.

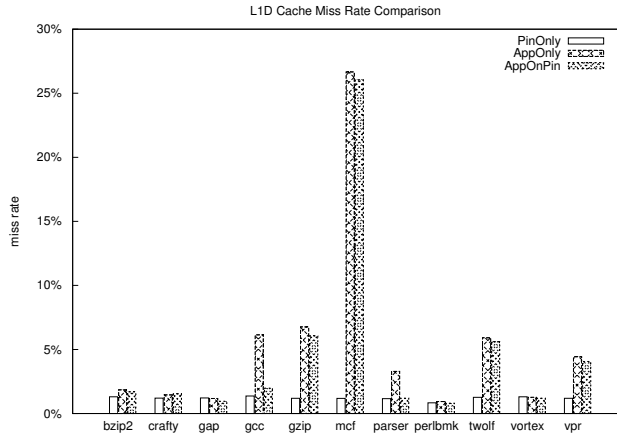


Figure 6. Miss rates of only Pin, only the application, and the application and Pin together, in the L1 data cache, collected from performance counters.

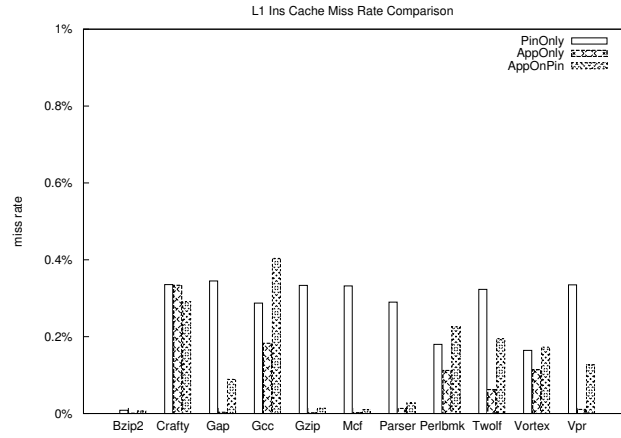


Figure 8. Miss rates of only Pin, only the application, and the application and Pin together, in the L1 instruction cache, collected from performance counters.

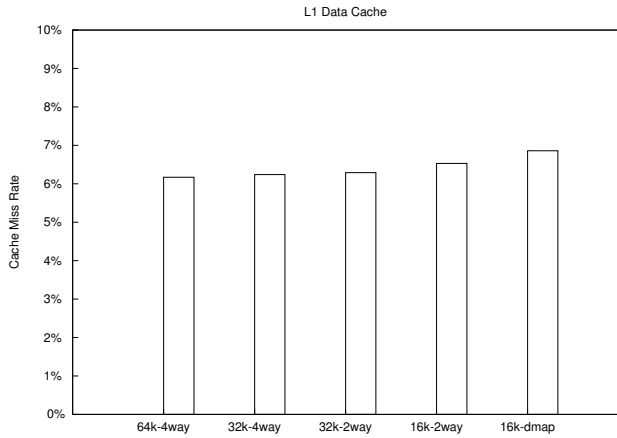


Figure 7. Pin's miss rate in the L1 data cache with respect to varying cache sizes, executing our 8-instruction program on SimpleScalar. This data largely represents startup and shutdown, which means a large number of the misses are unavoidable “cold start” misses.

3.3 Cache Performance

This section looks at the performance of the VEE in terms of the L1 instruction and data and L2 cache miss rate. Here, both the performance of the VEE and the performance of the application running under control of the VEE are important. This is because one design parameter is whether the VEE and application cores should share caches.

Figure 6 shows the miss rate in the L1 data cache of Pin, the application, and the application on Pin, collected from performance counters. The main thing to notice is that Pin has a fairly stable miss rate across all of the different applications, which suggests that its L1 data cache performance is not influenced as much by the input. Also interesting is that Pin, like several of the benchmarks, has a low miss rate, slightly more than 1%. This suggests a VEE-specific core may be able to use a smaller data cache without significantly affecting the performance.

To see how the miss rate is affected by varying the cache size, we then collected the miss rate from SimpleScalar-x86 for several

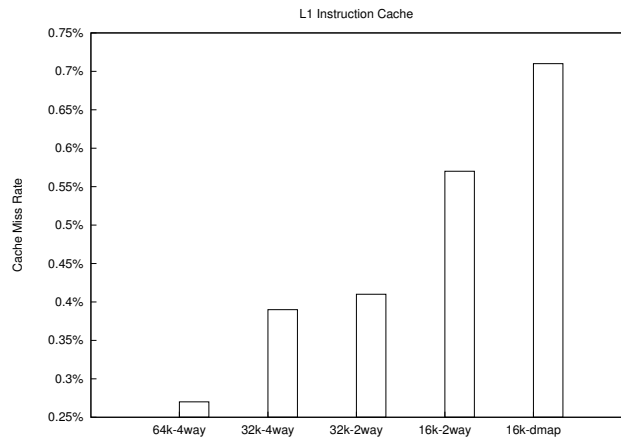


Figure 9. Miss rate of Pin in the L1 instruction cache with respect to varying cache sizes in SimpleScalar.

different cache configurations. The results are shown in Figure 7. The difference in rates reported by the performance counters and SimpleScalar can be attributed to the fact that it is primarily capturing startup and shutdown and thus has many “cold start” misses. The point to notice is that decreasing the cache size and associativity causes only a slight increase in the cache miss rate. This reinforces the idea that a VEE-specific core can have a smaller L1 data cache.

We now turn our attention to the L1 instruction cache. Figure 8 shows a comparison of the miss rates in the L1 instruction cache of Pin, the application, and the application on Pin. Unlike the L1 data cache, there is a much greater variation in performance across different inputs. Regardless, the average instruction cache miss rate of the VEE is no worse than the worst case for the benchmarks. Because of the potential for cache interference between Pin and the application, this provides a worst-case upper bound on the miss rate for running Pin on its own core—at worst, the VEE could jump back and forth between separate regions of code, which would have the same effect in terms of cache conflicts as sharing the core between the VEE and application. Thus, the cache size of current systems provides an upper bound on the cache size required for a VEE core.

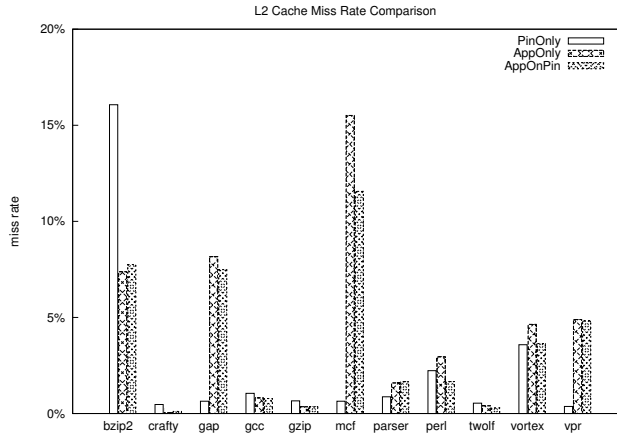


Figure 10. Miss rates of only Pin, only the application, and the application and Pin together, in the L2 cache, collected from performance counters.

We also considered the effect of varying L1 instruction cache sizes using SimpleScalar-x86, which is illustrated in Figure 9. Again we see that decreasing the size of the cache has a relatively small effect on the cache miss rate. Thus, just as with the L1 data cache, a VEE-specific core can have a smaller L1 instruction cache.

Finally, we look at the performance of the L2 cache. Figure 10 again shows a comparison between the miss rates of just Pin, just the application, and the application on Pin, in the L2 cache. Here we see Pin’s L2 miss rate varying wildly, from 16% on bzip2 to as little as 0.3% on vpr. The high miss rate for Pin on bzip2 appears to be an outlier and may be due more to conflicts from the application than Pin itself. This raises two questions, however: whether the compilation-based benchmarks (e.g., gcc or perl) show similar, wildly-varying behavior; and how the application and VEE may interfere with each other in the L2 cache.

To answer the first question, we present Figure 11, which shows a similar cache comparison broken out into the individual inputs of gcc and perl. In this figure, there are two things to notice. First, by comparing the second bar in each group (the application itself), we see that gcc has fairly stable performance across inputs, while perl’s miss rate varies from a negligible 0.00008% miss rate on input three, to a 5.5% miss rate on input seven. Second, we see that Pin’s performance varies not only with the application, but with the application’s input, from 0.6% to 1.6% on two inputs of gcc and 0.5% to 4.4% on perl.

The second question—how the application and VEE may interfere with each other in the L2 cache—is important because as shown in Figure 1, multicore architectures frequently share an L2 cache between cores. To begin answering this question, we present a quick approximation of the effect on the L2 cache performance of the application by running it under control of a VEE. To gather data about the cache performance of just the application while running under Pin’s control, we used Pin’s instrumentation interface to start the performance counters when exiting the VEE and stop the counters when reentering the VEE, thus collecting the miss rate of just the application executing from the code cache. These results are presented in Figure 12. This suggests that running under a VEE may improve cache performance in some cases; further evaluation is necessary to confirm and explain or overturn this result. For instance, the VEE’s interference in the L1 cache may be artificially inflating the number of accesses. Alternately, it may be due to changing the pattern of collisions in the cache—in this case, the VEE would still exert the same influence when running on a sep-

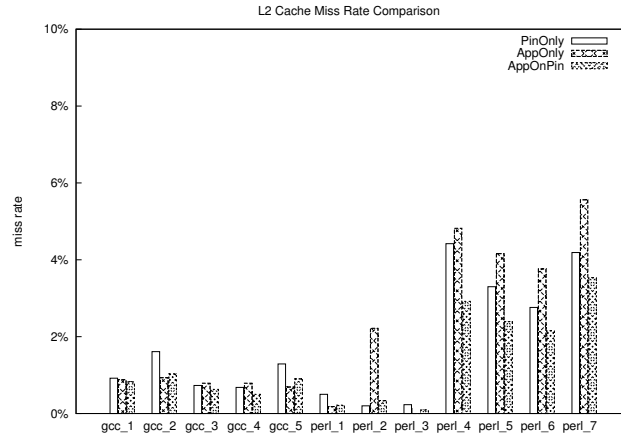


Figure 11. Miss rates of only Pin, only the application, and the application and Pin together, across the inputs of gcc and perl, collected from performance counters.

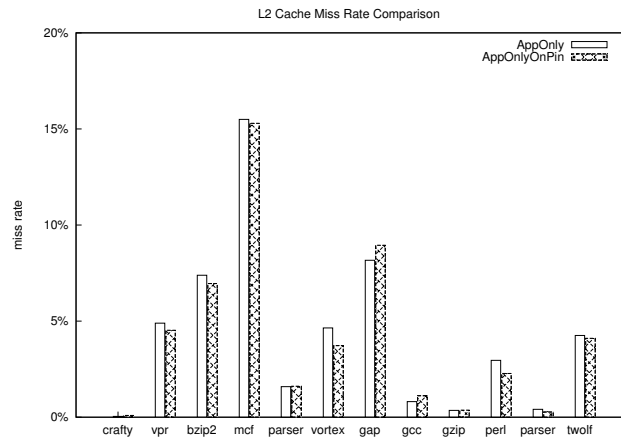


Figure 12. Miss rate of the application running natively and under Pin’s control, collected from performance counters.

arate core, as long as the cache was shared. Nevertheless, this is an interesting first step. A simulation that separated the L1 cache accesses of the VEE and application into two separate simulated caches would determine the overall impact of this factor.

We also performed a similar experiment on the effect on Pin’s miss rate of changing the L2 cache sizes on SimpleScalar-x86. However, due to the short runtime of our 8-instruction application and high number of cold start misses, the results were ultimately unhelpful.

3.4 Branch Prediction

We now turn our attention to the VEE’s characteristics with respect to the branch prediction hardware. First, we present a comparison of the number of dynamic branch instructions executed by Pin, the application, and the application on Pin, in Figure 13. The percentage of branch instructions varies across inputs, but this in itself is not a distinguishing characteristic—on any application, the different input sizes or the different paths taken on varying inputs could lead to such a variation. However, it is interesting in that the range of branch instruction percentages is equal to or greater than that of the benchmark applications. This would seem to suggest that

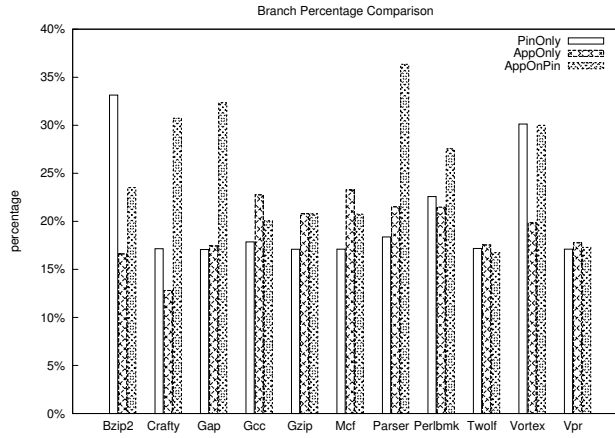


Figure 13. Percentage of branch instructions of only Pin, only the application, and the application on Pin, collected from performance counters.

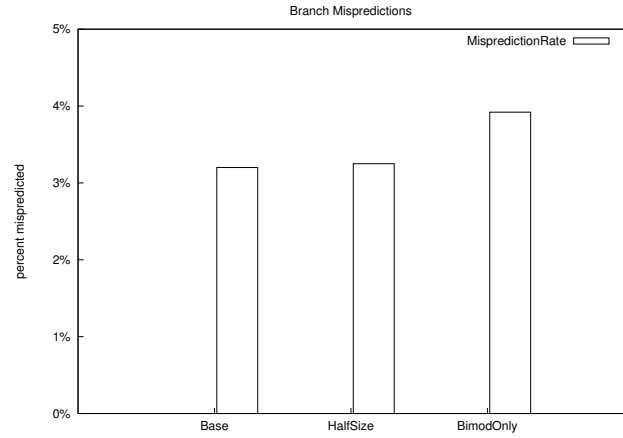


Figure 15. Percentage of branches mispredicted with respect to varying the branch predictor in SimpleScalar.

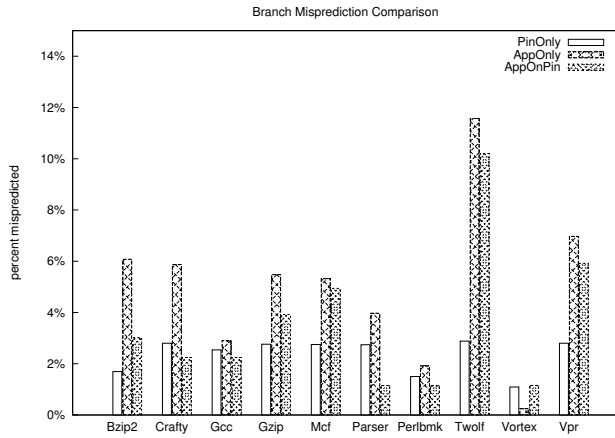


Figure 14. Percentage of branches mispredicted in Pin, the application, and the application on Pin, collected from performance counters.

the branch prediction hardware on a general-purpose core would also be a reasonable choice for a VEE-specific core.

However, the rate of branch misprediction is perhaps more interesting than just the raw numbers of branch instructions. In this vein, Figure 14 shows a similar comparison of the percentage of branches mispredicted. Here, we see that despite having a higher dynamic branch instruction count, the VEE tends to have a lower branch misprediction rate than the various applications. This may be the case if, for instance, the VEE has a lower number of unique branches, and the branch predictor is able to better capture entire patterns.

Figure 15 shows the effects on misprediction rate of varying the branch predictor configuration in SimpleScalar. In this figure, *Base* represents the baseline branch predictor configuration from Table 1, which is a combining branch predictor. *HalfSize* represents a combining predictor with half as much storage for branch histories, and *BimodOnly* represents only a bimodal predictor (*i.e.*, no 2-level predictor component). Here we see that decreasing the size of the branch history table has a negligible effect on the misprediction rate. Thus, a VEE-specific core should have smaller branch prediction structures.

3.5 Power Consumption

Based on the preceding characterization, we settle on a smaller core configuration with no floating point hardware, 16k direct-mapped L1 instruction and data caches, and a combining branch predictor with 8K entries (the *HalfSize* configuration). Because the VEE core shares a L2 cache with the application core, our VEE core configuration still reflects the 512K L2 cache.

Given this configuration, we compare the power consumption of executing Pin on the specialized configuration to executing Pin on our base configuration. To collect this data, we use Watch [8], modified to only count microarchitectural structure accesses when the VEE has instructions in flight. Clearly, decreasing the structure sizes will decrease their power consumption per cycle; however, if decreasing the sizes significantly impacts the execution time in cycles, the overall power consumption will be greater.

We tested this by running the benchmark applications under Pin’s control on SimpleScalar for a complete execution of the test inputs. For benchmarks with multiple inputs, we used the first input. Figure 16 shows the per-cycle power and total energy consumption due to the VEE on both our base configuration and our specialized VEE core, normalized to the power consumption of the specialized VEE core. Here we see roughly a 15% decrease in per-cycle power consumption, and up to a 5% decrease in total energy on applications except for *crafty*. The reason for this is unclear; since *crafty* on Pin has a higher L1 instruction cache miss rate than some of the other benchmarks, it may be due to instruction cache pressure. Thus, we have shown power savings in general due to using a customized core.

3.6 VEE Characterization Summary

In this section, we have described characterizations of Pin, a representative virtual execution environment, in terms of its floating-point hardware usage, branch predictor usage and misprediction rate, and miss rates in the instruction and data L1 and unified L2 cache. We have shown that a very small percentage of instructions are floating point, leading to the decision to remove floating-point hardware from the VEE core. The miss rates in the L1 caches are both low, and appear to suffer little degradation from decreasing the sizes of the caches. The L2 cache also has a low miss rate in general, although this may be affected by the guest application. There is also evidence that the VEE may positively impact the cache performance of the application, which validates the decision to share the L2 cache between the general-purpose and VEE cores.

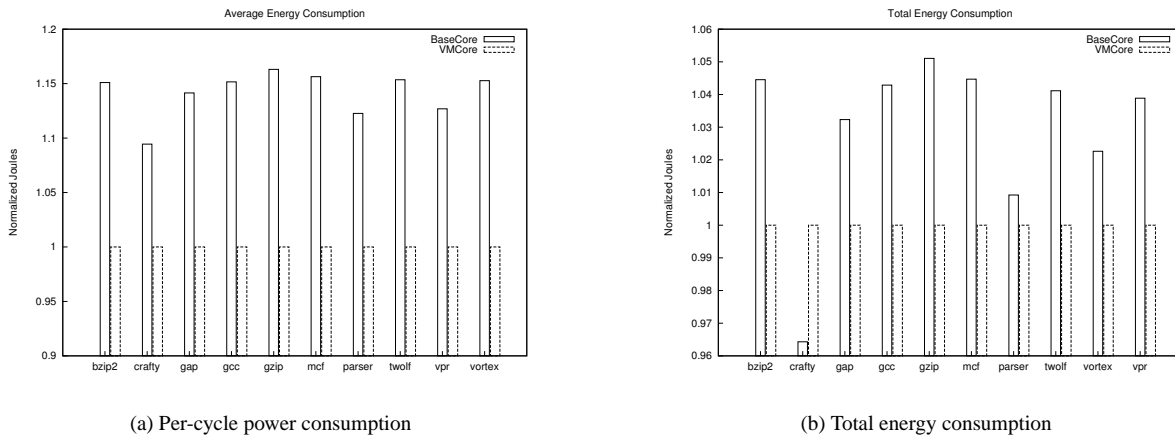


Figure 16. Comparison of per-cycle power and total energy consumption due to the VEE, on our baseline SimpleScalar configuration and our specialized VEE core.

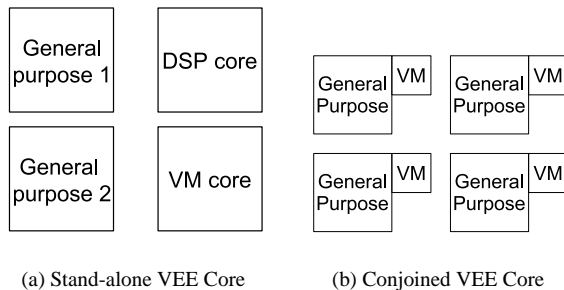


Figure 17. Overview of potential organizations of a heterogeneous CMP including one more more VEE cores.

Finally, we showed that although the VEE may have a higher-than-average percentage of branch instructions, it has a lower-than-average branch misprediction rate and does not suffer performance degradation from decreasing the size of branch history structures. Using these results, we chose a core design and simulated its power usage, showing that our specialized core saves up to 5% on power consumption over executing the VEE on the base configuration.

4. Chip Design for a VEE Core

Now that we have a design at the microarchitectural level, we turn our focus to design at the chip level. In this section we will discuss core layout options as well as extensions beyond a standard core to support executing the VEE on a separate core.

4.1 Core Layout

Figure 17 depicts two main classes of core layout. In Figure 17(a), all general-purpose cores share a single VEE core; in Figure 17(b), each general-purpose core is allocated its own VEE core.

Several factors play into the choice between these options. For instance, on a system primarily running process-level VEEs, a dedicated VEE core per general-purpose core would prevent different instances of the VEE from conflicting with each other. This also allows for sharing the floating-point hardware as suggested in Section 3.2. It also provides a short pathway for communicating com-

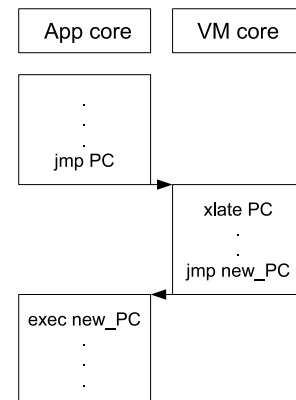


Figure 18. A general overview of the interaction between a VEE and its guest application. The application executes some amount of translated code until branching to a new address, at which point the VEE must take control, translate the new code into the code cache, and then resume the application’s execution in the cache.

piled code, as the code the VEE writes need only go to the shared L2 cache before being read into the application core’s L1 instruction cache. This may be a minor point, however; a microbenchmark which timed communication between cores on a quad-core processor suggested average communication times on the same order of magnitude for both shared and non-shared L2 caches.

The potential drawback to Figure 17(b)’s approach is low utilization. Ideally, the VEE executes briefly at the beginning of an application’s execution, after which execution is primarily the application running from the code cache. A single VEE core would have higher utilization, in the same way multiprogramming a general-purpose core yields a higher utilization, although it would lead to interference in the cache and branch predictor.

4.2 VEE Support

Figure 18 provides an overview of the run-time interaction between the VEE and its application. As discussed in Section 2.3, a context switch, which is the transfer of control and processor state from application to VEE and vice versa, is a source of non-trivial overhead as it requires saving and restoring the application’s state.

One positive side-effect of moving the VEE's execution to a separate core is that the application and VEE no longer need to share physical resources. In particular, the registers no longer need to be saved; in addition, contention in the L1 caches is reduced. This alone should lead to a reduction in overhead. Some transfer of information is still required; specifically, the application must let the VEE know where to begin compilation, and the VEE must let the application know where to resume execution. An additional reduction in execution overhead then would be add a simple channel between the cores to streamline this communication.

Running the VEE on a separate core opens further possibilities, such as parallelizing speculative compilation with application execution. In particular, some VEEs generate code by following fall-through paths until reaching an unconditional jump. In such systems, the VEE could speculatively generate code beginning at the target of the unconditional jump while the application executed previously-compiled code. In systems that build traces based on hot paths, such as DynamoRIO [10], a channel allowing the VEE to peek into the application core's branch predictor could be beneficial. Such speculation leads to the need for recovery mechanisms, such as the case where the application requests code other than that which the VEE is speculatively JITting. Previous work has addressed some of these issues [29, 39]; future work will continue to explore these design parameters.

4.3 Design Summary

This section has presented two categories of core layouts utilizing our specialized VEE core, then provided a discussion of additional support for the VEE that could be built into such a chip. Trade-offs involved include core utilization, interference, communication latency, and speculation recovery. Preliminary results suggest a single VEE core shared amongst general-purpose cores, with a mechanism to streamline communicating information between the VEE and applications.

5. Related Work

Some previous work has used hardware to benefit virtual execution environments. The ADORE system [11, 28] used hardware performance counters on the Itanium processor to profile execution and guide run-time optimizations. Zhang *et al.* presented the Trident framework [41], which added hardware structures to support their event-driven dynamic optimization system. They also seek to take advantage of hardware thread contexts to parallelize optimization with application execution, but do not seek to optimize separate cores for their optimization workload. ROAR [33] modified the architecture to support speculation and code motion in dynamic optimization systems while still allowing for precise exceptions in the context of the original application. Several codesigned virtual machines, such as Transmeta's Code Morphing Software [12], DAISY [14], and work by Kim and Smith [20], have addressed hardware support for virtual machines. However, none of the aforementioned works have fully addressed design of a core specifically for executing a VEE.

Recent processors have added ISA-level support for "classical virtualization" systems like Xen [7], including Intel VT [18] and AMD SVM [2]. These are designed to reduce virtualization overhead due to emulating privileged instructions, which must be monitored by the virtual machine monitor (VMM) to ensure isolation. Again, this is an example of an extension to existing hardware for virtual machine support, but does not approach VEE-specific design from the ground up.

A few projects have considered the design of a core based on the Java Virtual Machine (JVM), including JOP [35] and picoJava [31]. However, rather than designing a processor to efficiently execute the JVM, these machines approached the design to efficiently exe-

cute Java bytecode. The core design presented in our work should be equally applicable to running a JVM while still remaining flexible enough for other VEEs.

One final piece of work worth noting is a study on energy consumption and hardware behavior of virtual execution environments [15]. Their work, which focused on JikesRVM [3] and characteristics of the application and VEE together, can be viewed as complementary to our work.

6. Conclusions and Future Work

This paper has proposed the design of a core specific to virtual execution environments in a heterogeneous chip multiprocessor. We characterized Pin, a representative VEE, at the microarchitectural level, and used this characterization to guide our core design. Using hardware performance counters and simulation, we have demonstrated a difference between the VEE and several benchmark applications. Specifically, we have shown that only a small percentage of VEE instructions use floating-point hardware, and addressed ways to provide floating-point capability to the VEE without including hardware on its core; we have shown that a VEE core can have smaller caches without significantly impacting performance; and we have shown that a VEE core can have smaller branch prediction structures without significantly impacting performance. By reducing structure sizes, such a design was also shown to consume up to 5% less power. We have also presented an initial overview of design decisions at the core level of a CMP including one or more VEE cores, in terms of chip layout and support structures. Preliminary results suggest a single VEE core shared by multiple general-purpose cores, and the addition of communication channels to streamline interaction between the VEE and applications.

Follow-on work will include a characterization of other structures via simulation, such as functional unit occupancy or register file and reorder buffer size. A full multicore simulation will allow us to measure the utilization of a VEE core and interference of executing multiple instances of a VEE on a single core, leading to a more concrete design at the chip layout level. Finally, we will further explore the opportunities for improving VEE performance enabled by moving the VEE to a separate core.

References

- [1] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, December 2000.
- [2] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual, May 2005.
- [3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *15th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
- [4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, February 2002.
- [5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.
- [6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *32nd Annual Symposium on Computer Architecture*, pages 506–517, Madison, Wisconsin, June 2005.
- [7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of

- virtualization. In *SOSP '03: The 19th ACM symposium on Operating systems principles*, pages 164–177, Bolton Landing, NY, USA, 2003.
- [8] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Int'l Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, 2000.
- [9] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int'l Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
- [10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, California, March 2003.
- [11] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Symposium on Code generation and optimization*, pages 79–90, San Francisco, California, 2003.
- [12] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klamber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Symposium on Code generation and optimization*, pages 15–24, San Francisco, California, March 2003.
- [13] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.
- [14] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, Colorado, June 1997.
- [15] S. Hu and L. K. John. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 100–110, Ottawa, Ontario, Canada, 2006. ACM Press.
- [16] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 2–12, Ottawa, Ontario, Canada, 2006. ACM Press.
- [17] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multi-core architectures. In *2007 Conference on Design, Automation, and Test in Europe (DATE)*, Nice, France, April 2007.
- [18] Intel Corporation. Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture, April 2005.
- [19] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Order #253668-019, March 2006.
- [20] H.-S. Kim and J. E. Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.
- [21] S. T. King, G. W. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX'05: Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–15, Anaheim, CA, April 2005.
- [22] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, San Francisco, California, August 2002.
- [23] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.
- [24] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: 36th annual IEEE/ACM Symposium on Microarchitecture*, pages 81–93, San Diego, California, 2003.
- [25] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *37th annual IEEE/ACM Symposium on Microarchitecture*, pages 195–206, Portland, Oregon, 2004.
- [26] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: 15th conference on Parallel architectures and compilation techniques*, pages 23–32, Seattle, Washington, USA, 2006.
- [27] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: 31st symposium on computer architecture*, page 64, München, Germany, 2004.
- [28] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system, 2004.
- [29] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, Barcelona, Spain, November 2005.
- [30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, IL, USA, 2005.
- [31] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, 1998.
- [32] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2007.
- [33] E. M. Nystrom, R. D. Barnes, M. C. Merten, and W. mei W. Hwu. Code reordering and speculation support for dynamic optimization systems. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 163–174, Barcelona, Spain, 2001.
- [34] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: 7th conference on Architectural support for programming languages and operating systems*, pages 2–11, Cambridge, Massachusetts, United States, 1996.
- [35] M. Schoeberl. JOP: A Java optimized processor. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRRES 2003)*, pages 346–359, Catania, Italy, November 2003.
- [36] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *First International Symposium on Code Generation and Optimization*, pages 36–47, San Francisco, California, March 2003.
- [37] J. Sugeran, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *2002 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.
- [38] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: 22nd annual symposium on Computer architecture*, pages 392–403, S. Margherita Ligure, Italy, 1995.
- [39] D. Williams. Threaded software dynamic translation, 2005.
- [40] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS'07: Proceedings of the 7th International Symposium on Performance Analysis of Systems and Software*, San Jose, CA, April 2007.
- [41] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multi-threaded dynamic optimization framework. In *PACT '05: 14th Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, St. Louis, Missouri, 2005.