Constructing Virtual Architectures on a Tiled Processor[*]

David Wentzlaff and Anant Agarwal
CSAIL, Massachusetts Institute of Technology
Cambridge, MA 02139
{wentzlaf, agarwal}@cag.csail.mit.edu

Abstract

As the amount of available silicon resources on one chip increases, we have seen the advent of ever increasing parallel resources integrated on-chip. Many architectures use these resources as individually controllable, parallel processing elements. While such architectures excel at parallel applications, they seldom support legacy single-threaded applications. In this work, we propose using parallel resources to facilitate execution of legacy codes with acceptable performance on parallel architectures containing a drastically different instruction set through the use of an all software parallel dynamic binary translation engine. This engine spatially implements different portions of a superscalar processor across distinct parallel elements thus exploiting the pipeline parallelism inherent in a superscalar. This virtual microarchitecture facilitates changing the allocation of silicon resources between different superscalar units in software which is not possible when special purpose physical resources are built. We propose building dynamically reconfigurable architectures that inspect the current virtual machine configuration along with the dynamic instruction stream and change the configuration to best suit the program's needs at runtime. An x86 to Raw parallel translation engine was built in which tiles dedicated to translation can be traded for tiles dedicated to the memory system as an example of dynamic reconfiguration.

1 Introduction

As we look to the future, trends suggest that we will see a proliferation of on-chip, distributed parallel processors such as tiled processors [20, 16], multi-core processors [1], and chip multi-processors [22]. Building such processors is an effective manner to utilize the ever increasing silicon resources afforded to the computer architect. While these parallel architectures provide for large performance improvements over typical sequential processors, especially on parallel applications, they typically do nothing to accelerate existing sequential binary applications. In many cases, due to architectural mismatches such as differing instruction sets, lack of a MMU, non-coherent memory, differing I/O interfaces, and lack of OS support, future on-chip parallel architectures may not be able to run our current set of application binaries directly. One solution is to either recompile or recode all of the applications in the world to take advantage of these new architectures. Unfortunately, this requires reverification of all of the applications that are currently used, and platform porting may not even be possible to some of the architectures due to differing memory models and lack of OS support. Even if feasible, this porting and verification effort would be a huge amount of duplicated effort. This leaves designers of on-chip parallel architectures in a bind; they want to focus on the exciting new parallel applications that their architectures excel at, but they still would like to maintain compatibility with the industry-standard suite of applications.

Ultimately, we would not like to see the creativity of designers of future architectures impeded by the requirement of backward compatibility, yet it is also a suboptimal solution to require all software be redesigned for these future architectures. In the future, legacy ISAs such as *x86 will no longer simply be an ISA, but rather x86 and the accompanying ecosystem will become an application* that all future architectures will have to execute effectively. One solution may be to use a sequential translation system to emulate a legacy architecture to maintain compatibility. Unfortunately, this solution may not provide sufficient performance on legacy codes and does not utilize any of the parallel resources provided by future parallel architectures which may be the primary growth path in the future. *In this work, we investigate using parallel resources in a tiled processor environment as an enabling technology to accelerate emulation.* We introduce several new mechanisms of exploiting parallelism in a tiled processor to accelerate the execution of a single threaded legacy application.

In this work, we have designed and implemented a parallel dynamic binary translation engine for a tiled architecture. While much previous work has gone into dy-

[*]Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO-4), New York, NY, March, 2006.

namic translation for a parallel architecture, typically a VLIW [9, 11, 12], in this research we take a different focus and utilize other forms of parallelism than simply scheduling translated code to a VLIW processor. The novel mechanisms presented in this paper that allow the utilization of parallel resources to accelerate cross platform binary execution are:

1. Speculative Parallel Translation
2. Spatial Pipeline Parallelism
3. Static and Dynamic Virtual Architecture Reconfiguration

These techniques focus on using the tiled processor as an ASIC or FPGA-like fabric that has customized functional units. To that end, this work can be thought of as implementing a virtual superscalar microarchitecture across a tiled processor fabric. We did not restrict ourselves to faithfully implementing a pre-existing processor design. Rather, we took the techniques embodied in superscalar design, which are effective in exploiting transistor parallelism, and applied them when it made good engineering sense to do so. An example of this FPGA-like design is the fact that we explicitly manage on-chip layout and communication distance.

One of the key portions of any dynamic translation system is the translator itself. Unfortunately, one does not want to incur the cost of translation on the critical path of your computation. To solve this problem, we introduce speculative parallel translation. Speculative parallel translation traverses a program's possible execution paths and translates them before the piece of code is needed. This removes the cost of translation from the execution of any program because the translation cost of future basic blocks is overlapped with the execution of the current block. We believe that this technique can even be applied to superscalars in the form of more aggressive decoding into a larger trace-cache or code cache like structure.

Spatial pipeline parallelism is the notion of coarsely pipelining needed computations across neighboring tiles. We exploit this form of parallelism by pipelining our memory system and pipelining code cache accesses. Lastly we introduce static and dynamic virtual architecture reconfiguration. Static reconfiguration is motivated by the fact that different programs have different requirements in terms of working set size, amount of ILP, and amount of instruction bandwidth. On a non-virtual processor, the architect needs to choose one configuration of all of these parameters at design time, while in a virtual environment, different machine configurations can be chosen to fit a particular program's needs. This can even be extended for use inside of a program, assuming that a program has phases, and is called dynamic virtual architecture reconfiguration. Dynamic reconfiguration allows the emulator to inspect itself along with the programs needs to rebalance silicon resources at runtime.

To explore these concepts, we have built a prototype parallel dynamic translation system. The prototype system uses x86 Linux as the guest ISA and operating environment and executes on a Raw tiled processor host. We present a fully functional system that executes arbitrary, unmodified, userland statically-linked Linux x86 binaries on the Raw prototype chip. The Raw host architecture is significantly different than the guest x86 architecture. Namely a vastly different ISA, lack of memory translation, lack of protection, lack of condition codes, and lack of a hardware instruction cache on Raw are some of the challenges that this design faced and attacked in an all software dynamic translation environment. To mitigate this mismatch in architectures, this work exploits the parallel resources found on the Raw processor to accelerate binary translation.

All of the results presented in this paper were collected on actual Pentium III and Raw hardware. No modifications were made to the x86 binaries, the Pentium III hardware, or the Raw hardware. Additions to the Raw hardware would have improved the performance of Raw running x86 binaries, but in this study we have focused on pushing the limits of an all software approach. This work leaves the investigation of enhancing the hardware in tiled processors for the purpose of accelerating emulation to future work. We evaluated our parallel dynamic translation system across the SpecInt 2000 benchmark suite and found that our system with software memory translation attains approximately a 7x-110x slowdown when x86 binaries are run on Raw compared by cycle counts against a Pentium III.

This paper is organized as follows. Section 2 examines how the parallel resources of a tiled processor can be exploited to accelerate dynamic binary translation. In Section 3 we describe the system implementation and tradeoffs. We present and discuss the results of several differing virtual machine configurations in Section 4. Sections 5 and 6, respectively, present future and related work. And finally we conclude.

2 Exploiting Parallelism

2.1 Speculative Parallel Translation

Dynamic binary translators need to translate code from one architecture to another architecture at runtime. This can be quite expensive, especially for applications that either have a very short runtime or contain a large number of instructions that are executed infrequently. For long running programs that execute the same set of instructions frequently, techniques such as a code cache can amortize the cost of translation over many executions of a translated block. On sequential architectures, translation still uses valuable cycles that could otherwise be spent running operations from the program. We propose a better solution for translation. Instead of stealing cycles away from the main program's execution thread to do translation, specula-

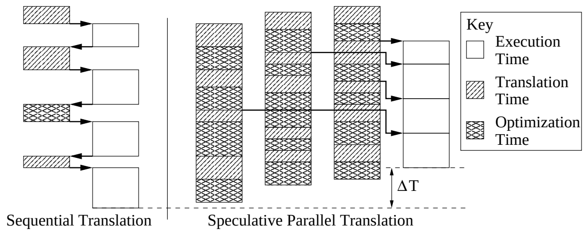

Figure 1. Example Speculative Parallel Translation. Time increases from top to bottom.

tive parallel translation utilizes parallel execution resources to translate the program in the background. With this approach, when the main execution thread reaches a portion of the program it has not previously executed, instead of stalling waiting for translation of that code, the speculatively translated code is simply recalled and executed.

Figure 1 illustrates the advantages of speculative parallel translation. Shown on the left, is a hypothetical sequential translator. The processing element alternates between translating and executing the program of interest. Along the way, the translator decides that it is wise to optimize a block, indicated by the cross-hashed region. On the right side, a speculative parallel translator is shown running on four processor cores. In the speculative parallel translation example, three processors are used for translation and optimization, while one is dedicated to execute the translated code. Because the translation and optimization cores run ahead translating the program, the execution core is able to execute the translated code without the translation delay seen in the sequential case. By removing the translation time from the critical path, the speculative parallel translation example is capable of completing earlier than the sequential example as denoted by ΔT.

One challenge that exists with speculative parallel translation is determining which code is most likely to be needed. A naive implementation is simply to traverse the program graph in control flow order. When a branch is reached, the translator spawns a new translation thread for each path that the program may go. We call this approach speculative parallel translation because the translator is doing speculative work, translating portions of the program that may or may not ever be executed. While it is possible that a large amount of the work that is done may not be needed with this scheme, these parallel resources still contribute to accelerating the main thread's execution.

One way to mitigate translating unneeded code sections is to use some form of prediction when a branch is reached to prioritize what is to be translated. Also, proper prioritization may actually accelerate the main thread's execution by not scheduling portions of code to be translated that may take away resources from sections of code that are critical to the program execution. Unfortunately this form of prioritization and prediction is difficult. It is effectively the same problem as constructing a branch predictor with no previous branch information. Typically branch predictors use history information to determine which direction a branch transitions control to. In this case, the translator is translating code before it is even executed, and hence can be thought of as a first touch branch predictor. Some static heuristics can be applied in this situation such as predicting backward branches taken. Ball and Larus suggest other static heuristics for branch prediction in [3].

In this work we used simplistic static branch prediction along with a prioritized set of queues to determine what address should be translated next. The different levels of priority are used to determine which address should be translated when a tile becomes free. When enqueuing, the priority is determined by the depth the current block is from a piece of code that is known to be on the correct execution path of the program. Thus as the work becomes more speculative, or further from the last know piece of executed code, it is given a lower priority. The results of speculative translations are stored in a large code cache until they are needed.

This procedure works for direct branches, but does not handle register indirect branches well. For indirect branches, the translator is not able to determine what address is the next appropriate address that the program may branch to until runtime. Typically, indirect branches come in the form of either function returns or calls through a jump table. For call returns, it is typically possible to determine the return address at call time. We use a return predictor which adds the address after a call instruction onto a low priority translation queue. The return address is put on a low priority queue because the code inside of the function has a higher probability of being needed then the return location. For jump tables, without symbol information, it is effectively impossible to know what address may be called. Currently our system does not speculatively translate beyond unresolvable register indirect jumps.

Speculative parallel translation can be applied to other forms of translation. For instance, most modern processors that execute x86 code translate or decode instruction streams into micro-operations that are later executed. They use specialized hardware to perform this translation and store a small number of decoded instructions into a trace cache. Another approach is to translate more aggressively as described in this section and cache the translations in a much larger cache. This cache may possibly reside in main memory or on some physically distant portion of the microprocessor. Ultimately this may reduce energy usage because it would prevent re-decoding previously decoded instructions. Also, instead of performing this decoding with special purpose hardware, this could be performed by general purpose processors that may be reallocated once the working set of instructions have been translated. This can be applied to architectures such as Transmeta's. Instead of steal-

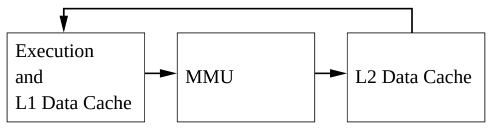

Figure 2. Pipelined Memory System.

ing cycles on the main computation unit, Transmeta could use speculative parallel translation on an array of small processors situated on a physically distant portion of the silicon. This would reduce translation cost significantly, but might be at the cost of energy and code cache memory.

Lastly, speculative parallel translation changes the conventional wisdom in dynamic translators. Typically dynamic translators employ some form of hot spot optimization. Hot spot optimization detects the most frequently used or "hot" portions of a program and only optimizes those portions. This prevents stealing precious execution cycles from the main thread to optimize code that is seldomly run. With speculative parallel translation, because the translation is removed from the critical path of the program's execution, expensive optimizations can be applied at translation time and not steal cycles from the main thread. Rather, the translated code that is run is now optimized, thus reducing the time required for it to execute. We still believe that hot spot analysis can improve performance by not wastefully optimizing code, but the cost of optimizing code is lower in a parallel environment. In this project we decided to leave full optimizations turned on for all blocks because the cost of optimization, which was off of the critical path, was outweighed by the benefit of executing optimized code.

2.2 Spatial Pipeline Parallelism

Tiled processors can be thought of as a substrate for circuits to be implemented on top of. One such circuit is that of a processor. We propose exploiting a tiled architecture as an ASIC or FPGA. Typically FPGA and ASIC designs exploit multiple forms of parallelism. One form of parallelism that can be easily exploited is pipeline parallelism. This allows for logic to be used sequentially by passing data between fixed function stages. In a dynamic translator there are many functions that need to be completed that may not exhibit thread or data-level parallelism. Thus turning to pipeline parallelism allows for the exploitation of parallelism in a different manner. Pipelining can also increase the throughput of a needed resource. For instance, in this work's prototype system, the memory system was built out of multiple tiles in a pipelined manner. When an access is made to the cache, it is first passed to the MMU unit for translation and then onto the cache tile as shown in Figure 2. While the memory request is being serviced, the main execution processor is free to execute other non-dependent work and issue further memory requests. Like all modern FPGA and ASIC designs, wire delay is a significant factor. This is also the case when pipelining across tiles, thus spatial pipelining takes into account wire delays to minimize latencies.

This work is a proof of concept that pipelining a virtual processor across a substrate of tiled processors is feasible. This idea can be extended to an extreme by implementing a out-of-order superscalar across many tens of tiles. Sets of tiles could be ganged together to implement the front end translation, after which the instructions pass to a set of scheduling tiles. Then the instructions can be passed to processors that perform renaming and then reservation station tiles that issue instructions to multiple execution units. Finally the instruction results could be passed along to a set of tiles that implement a reorder buffer in software for instruction retirement. This form of pipelining would effectively allow a parallel processor to be used to speed up the execution of sequential codes.

2.3 Static and Dynamic Virtual Architecture Reconfiguration

One of the major design decisions that any computer architect grapples with is how to provision the silicon area that he or she has to work with. Even with a relatively small number of knobs that can be tweaked, there exists a large, exponential, number of different designs that can be created. With all of these different possible designs, the architect ultimately chooses some configuration of the resources and encodes this into circuits on silicon. Typically the architect takes a representative cross section of the applications that the chip will execute and optimizes parameters such as cache size, fetch bandwidth, bandwidth to main memory, number of functional units, number of physical registers, etc. The parameters chosen may be optimal across the benchmark suite, but it is probable that they are not optimal for any one benchmark but rather are a compromise. With a virtual architecture like the one presented in this paper, the architect does not need to determine the layout and allocation of silicon resources at chip design time. Different virtual architectures can be created and tailored to best suit a particular application.

We propose and demonstrate static virtual architecture reconfiguration. With static virtual architecture reconfiguration, there are many differing virtual architectures that are implemented on top of a substrate. In this work the substrate is a tiled processor, but this may also be a multi-cored processor. The configuration of the virtual architecture determines the relative amount of silicon resources dedicated to any one function. This reconfigurability frees the designer from designing only one architecture determined at fabrication time. Reconfiguration comes at the cost of requiring all of the physical silicon resources being able to perform, possibly through some form of software emulation, all functions that a normal processor would perform. This requirement is a good match for homogeneous tiled processors which allow any piece of silicon to perform any function modulo some emulation cost.

We can take this idea of silicon reconfiguration one step further and apply the same techniques dynamically. Within any given application, it may be the case that at different portions of the program a different allocation of the silicon resources may be optimal. This idea is motivated by the insight that programs typically transition through multiple phases throughout their runtime. When a program begins, the program has not been translated or decoded yet, thus most of the silicon resources should be dedicated to translation. After a significant portion of the program has been translated, the program may need more functional units because it has reached a highly parallel portion of its execution. Then the program needs to access memory often to store the computed results. With dynamic reconfiguration, the virtual architecture detects this situation and allocates more resources for cache. Finally the program reaches an indirect branch and the program requires more translation, and resources are allocated to that need. With dynamic architecture reconfiguration, there is some centralized manager that must make decisions when to reconfigure. This manager introspectively analyzes the current configuration of the virtual machine, the dynamic instruction stream, and the needs of the dynamic instruction stream. Because this level of optimization is occurring at runtime, all information is known which allows the reconfiguration manager to take advantage of quantities that only can be determined accurately at runtime. The dynamic reconfiguration manager can use information such as cache (instruction and data) miss rates, the dynamic determination of actual instruction level parallelism (typically not determinable at compile time due to address aliasing), the number of basic blocks waiting to be translated, the dynamic bandwidth needed to differing levels of cache hierarchy, and any other dynamically introspective meta-data about the program.

Dynamic reconfigurability does come with some cost. When changing the virtual machine configuration, there is a cost associated with reconfiguration and a cost to monitoring. To mitigate the cost of monitoring, sampling of the reconfiguration metrics can be employed to make the monitoring cost inconsequential. An example of reconfiguration cost is the cost associated with changing the size of the L2 data cache. When the L2 cache physically changes size, the contents of the L2 cache need to be flushed and written back to main memory. Events like flushing of a cache can be quite expensive if they occur often, thus any type of reconfiguration system should have hysteresis built into the system to prevent too frequent reconfigurations. In this work we prototyped dynamic reconfiguration by dynamically trading off the number of L2 data cache tiles against the number of translation tiles. To determine when to reconfigure, the length of the work queues of blocks to be translated was used. This dynamic reconfiguration allowed our translation system to beat the best statically determined configuration on some benchmarks thus demonstrating the power of introspective dynamic reconfiguration.

3 System Description

3.1 Background

Before this paper examines the implementation of an x86 dynamic translator on the Raw processor, the details of the Raw processor and x86 instruction set need to be discussed. The Raw processor is a general purpose tiled processor. It consists of 16 identical MIPS-like processors arranged in a 4x4 grid integrated on one die. These 16 processor cores are tied together by four first-class register mapped communication networks. Two of the networks are routed dynamic networks and two are software routed static switch networks. Each tile also contains 32KB of hardware managed data-cache, 32KB of software managed instruction memory, and 64KB of software managed switch instruction memory.

A tile is the basic repeated structure that the Raw processor is built out of. Inside of a Raw tile, there is an 8-stage 32-bit processor pipeline. The ISA for each tile's main processor is derived from the MIPS ISA. The 16 Raw tiles share global off-chip memory but there exists no support for cache coherent shared memory. Also, Raw lacks an MMU or any form of memory protection.

The guest architecture that we are emulating is the industry standard x86 or IA-32 architecture. The original x86 ISA is relatively well organized for an accumulator based machine, but has had many additions that now make it a quite complicated instruction set with many subtle nuances in how each instruction operates. x86 is primarily a CISC instruction set which uses condition codes/flags to make branching decisions. Typically, every ALU operation sets some subset of the global flag state, and most operations can touch memory. x86 is a two operand instruction set with a variable length encoding which makes decoding the full instruction set quite challenging. Finally, modern x86 code makes use of a virtual memory system along with a protection schema.

3.2 Design Overview

Our emulator is similar to most best-of-breed dynamic binary translation systems, and leverages much of the previous work on translators such as dynamic binary translation, making use of a code cache, and chaining branches in the last level of code cache whenever possible. But, many design trade-offs change when they are considered in a distributed environment. One example is that the cost to optimize blocks of translated code is less than in a typical translation system. In a parallel environment, optimization is done in the background on processing resources that might otherwise be left idle, while in a sequential translator, any

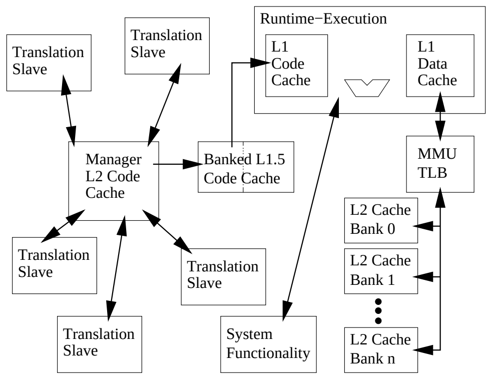

Figure 3. Partitioned Block-Level View of the Translation System

processor time that is used to do an optimization steals cycles from the critical path of the program.

Figure 3 presents a block-level view of the translation system. Each box represents a tile, processor core, executing a portion of the emulation system. The runtime-execution tile contains the runtime engine, the L1 code cache, and the L1 data cache. This tile is responsible for executing all of the translated code and executes the primary dispatch loop along with maintaining the lowest level of the code cache.

To service instruction misses in the L1 code cache, the banked L1.5 code cache is consulted. The L1.5 code cache utilizes two tiles as a local cache of already translated code for quick access. If requested code is not in the L1.5 code cache, the code request reaches the manager L2 code cache tile. The manager tile is responsible for maintaining the code cache that lives in main memory and for coordinating the parallel speculative translation units. The L2 code cache is 105MB and is stored in off-chip DRAM. There are many translation slave tiles which run ahead and speculatively translate possibly needed code.

Translation and optimization occurs on the translation slave tiles. The first stage of the translator leverages a small portion of the Valgrind memory debugger [15]. Valgrind was designed to be a memory debugger and uses just-in-time code instrumentation technology to detect memory leaks. The parsing is implemented as several large switch blocks. With the exception of this parsing and some basic high level code cleanup routines, all of the translator was custom built for this project. The translation slaves do code generation from a x86-like intermediate representation to a low level MIPS-like IR. Many standard compiler optimizations are applied at this level and finally the translated code is deposited in the L2 code cache.

When the runtime-execution tile needs to use more memory that is stored in the in-tile L1 data-cache, it requests data from the emulators pipelined memory system. The first step along this path is the MMU and TLB tile. This tile is responsible for translating x86 virtual to x86 physical addresses. It is also responsible for translating x86 physical to Raw physical addresses. The MMU tile then farms the memory request out to one or more L2 cache tiles. The L2 cache tiles are set up in a transactor style that service memory requests for fractions of the physical address space. Lastly the system contains a tile dedicated to servicing system call requests.

4 Results and Analysis

4.1 Methodology

In this work, we focus on evaluating a prototype parallel dynamic binary translation system built on top of the Raw tiled processor as a proof of concept for several ideas about constructing virtual architectures on parallel systems. We strived for realism wherever possible and as such all of the numbers presented are gathered on real hardware. No simulations were used and hence no modification to any of the hardware resources in either the Pentium III or the Raw processor was done. Gathering test results on existing hardware is a huge win with respect to the time taken to run benchmarks, and allowed us both to gather result data for larger input sets and to gather significantly more datapoints than otherwise would have been feasible in a simulation environment. Running on real verified hardware also lends credibility to the legitimacy of such results in a real world environment, but it makes the engineering effort higher because the hardware implementations are not modifiable thus this work has to accept them with all of their blessings as well as their faults.

Throughout this section, we use performance on the SpecInt 2000 benchmark suite as the metric of comparison. This benchmark suite was compiled using GCC 3.0.4 with the “-O3” optimization flag. All of the binaries were statically linked and use newlib 1.9.0 as the standard ‘C’ library. The C++ benchmark 252.eon was omitted from these results because we were unable to build and statically link libstdc++ with newlib, and currently our dynamic translation system is only able to handle statically linked binaries. The parallel translator in this paper currently does not support Intel’s x87 floating point arithmetic. SpecInt applications primarily use integer arithmetic, but surprisingly, as their name does not imply, include a small amount of floating point operations. Thus the applications were compiled with the gcc flags “-msoft-float -mno-fp-ret-in-387” which use a soft float library available inside of libgcc when floating point math is required. The exact same set of binaries was run unmodified on both the Pentium III and the translator running on Raw. The large datasets from MinneSPEC [13] were used throughout this paper.

To compare across platforms, a clock-for-clock comparison methodology was used. Thus the period of one Pentium III clock cycle was considered to be equivalent to one clock

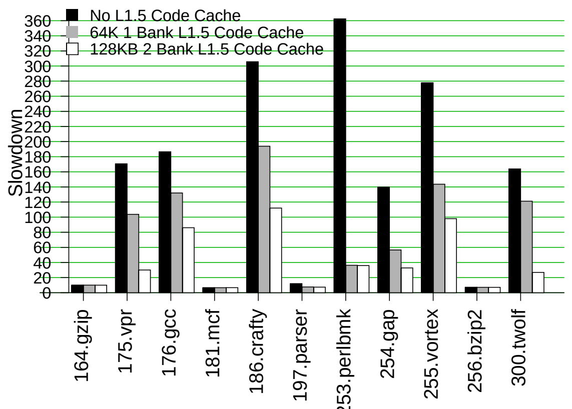

Figure 4. Comparison of L1.5 Code Cache Sizes

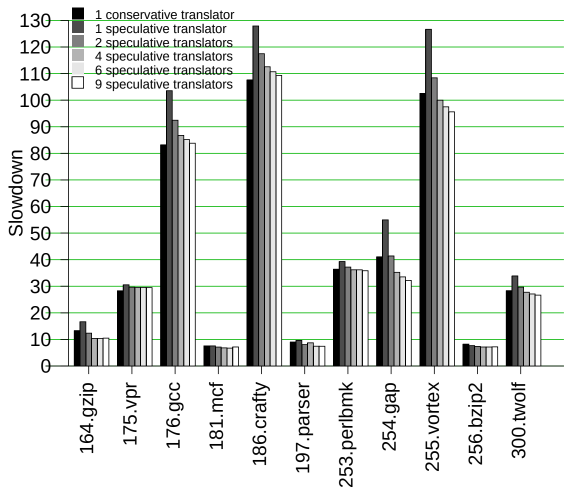

Figure 5. Comparison with Differing Numbers of Translation Tiles

cycle on the Raw processor. We believe this to be a fair comparison if both of these designs were to be implemented in similar design styles with similar design efforts. A detailed clock normalized and non-clock normalized comparison of these two architectures can be found in [21]. Unless otherwise stated, all of the numbers in this paper are presented as slowdown when compared to a Pentium III and can be calculated as: $\frac{CyclesOnTranslator}{CyclesOnPentiumIII}$.

4.2 Code Cache Sensitivity

In this work, we investigate full applications versus focusing on kernels. Large instruction working sets is one of the drawbacks of utilizing sizable applications such as those found in SpecInt. Applications particularly stress translation systems if their working set of instructions is larger than the translator's lowest level of code cache. We begin our results with an investigation of our system's sensitivity to code size.

In our initial design, we did not have a L1.5 code cache, but we noticed that some of the applications' instruction working set size was larger than the level 1 code cache available in the runtime-execution tile. This caused severe performance degradations on certain benchmarks, and persists even with the L1.5 code cache. The addition of a L1.5 code cache is an example of how parallel resources that were not otherwise being productively used can be reallocated to act as caches and hence speedup program execution. Figure 4 contains results from three differing machine configurations. One configuration has no L1.5 code cache, one contains a 64K code cache which required the dedication of one tile, and the last utilizes two tiles as a 128KB code cache.

As can be seen, vpr, gcc, crafty, perlbmk, gap, vortex, and twolf all contain code working sets larger than the L1 code cache of the translator. The L1.5 code cache has a longer latency than accessing the L1 code cache, and it prevents chaining. We believe that it may be possible to use a slightly better L1 code caching algorithm than the currently employed tight packing and flushing algorithm, but ultimately these benchmarks are capacity limited in the L1 code cache. With a larger L1 code cache or the addition of a true hardware Icache system in Raw, performance could be significantly improved for the benchmarks that exceed the L1 code cache capacity. A hardware Icache would help by allowing a larger virtual L1 code cache to be used than fits on-chip, and chaining could be done across this virtual cache. In the current system, chaining can only occur once code is copied into the instruction memory of the execution-runtime tile because it is only at this point that the absolute position of the relocatable code block is known.

4.3 Speculative Parallel Translation

To investigate the virtues of speculative parallel translation we ran the SpecInt benchmark suite with differing numbers of translation units. We ran this experiment with 1, 2, 4, 6, and 9 slave translators. For the one translator case, we performed these tests with and without speculation. For the non-speculative (conservative) case, the one translator did not translate ahead in the program and was always ready waiting for a L2 code cache miss to occur. In all of the other cases, if a translation request arrives from the runtime-execution engine and all of the slave tiles were being used, the request would have to wait. The non-speculative, conservative case approximates the translation portion of a classic sequential translator.

Figure 5 shows the results for this experiment. As can be seen, with the exception of vpr, gcc, and crafty, using speculative parallel translation accelerates the program execution over the conservative case. Also the trend shows that as more translation resources are added, the applica-

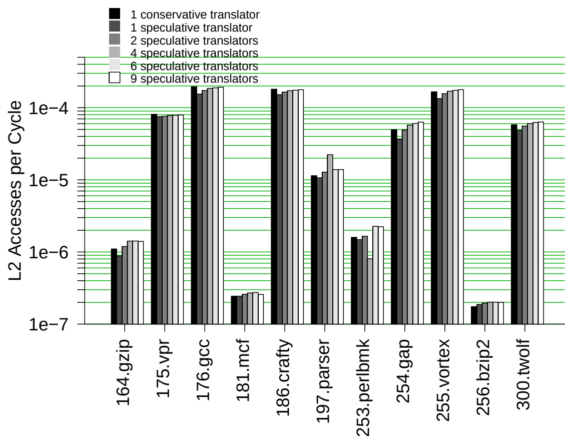

Figure 6. Number of L2 Code Cache Accesses per Cycle

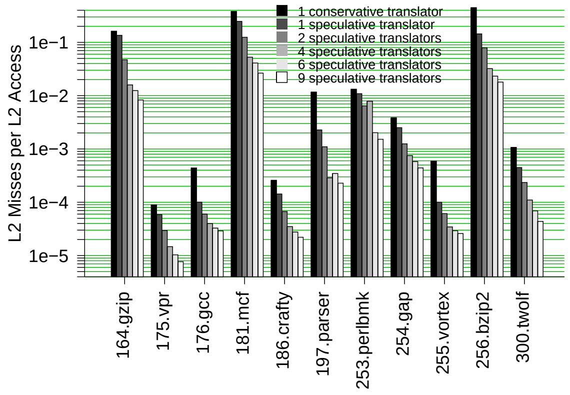

Figure 7. Number of L2 Code Cache Misses per L2 Code Cache Access

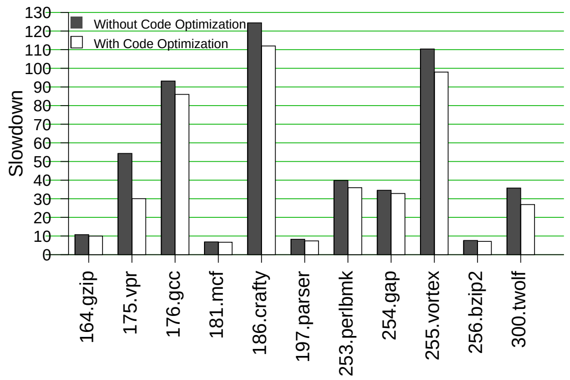

Figure 8. Comparison of No Code Optimization versus Code Optimization

tions execute faster. Lower bars indicate lower slowdown and faster execution. The 9 translator datapoint trades off three L2 data cache tiles for three extra translators over the 6 translator datapoint. Hence in some of the memory intensive applications the 9 translator version was slightly slower than the 6 translator case.

We were a little surprised to see that for vpr, gcc, and crafty, the parallel translation configurations were actually slower than the conservative single thread translation case. We believe that this occurs for two reasons. First, in our implementation of speculative parallel translation, we do not use a preemption model. Thus if a request comes in from the execution engine for a particular piece of code that is not in the L2 code cache and all of the translation slave tiles are currently occupied, this request stalls until a slave finishes its current piece of work. We believe that this can be mitigated by reserving a slave tile for these requests to reduce latency for critical translations or by adopting a preemption mode. Second, the L2 code cache and manager tile is a shared resource that all of the slave translators use to store their results in. This causes significant traffic to and from this central resource, and in turn delays sending and receiving data to the runtime-execution engine. To investigate this further, we plotted the rate at which a particular program accesses the L2 code cache per cycle of execution time. Figure 6 shows these results. The rates at which these applications access the L2 code cache vary over three decades. Our hypothesis about congestion at the L2 code cache was confirmed by this experiment because vpr, gcc, and crafty all experience a high rate of accesses to the L2 code cache. In the future, banking or decentralizing the L2 code cache will mitigate these problems.

The rate at which these programs miss in the L2 code cache, and hence have to be translated can be seen in Figure 7. This graph confirms that as more speculative threads are added, the miss rate in the L2 code cache decreases. This is encouraging for speculative parallel translation being able to decrease the critical path for translation as even more translation resources are added, and hence speedup programs that otherwise have high translation occupancy due to poor code locality.

In addition to parallel translation, we wanted to investigate whether code optimization on every block was worth the added occupancy of performing the optimizations. Figure 8 shows the runtime of the SpecInt benchmark suite with and without optimization during translation. For all of the benchmarks, the occupancy of performing optimization in a speculative parallel environment was far outweighed by the decrease in runtimes afforded by the optimizations. For these runs, a dynamically reconfiguring (6 to 9 slave translation tiles) configuration was used.

4.4 Static and Dynamic Virtual Architecture Reconfiguration

We investigate the tradoffs involved in virtual architecture reconfiguration in this section. Figure 9 contains the runtimes of benchmarks run with five different architecture

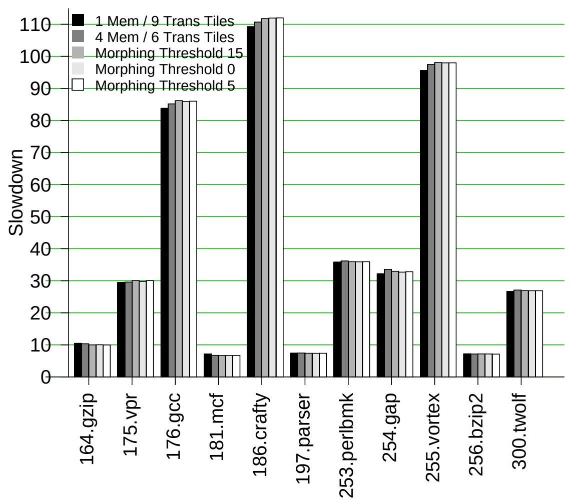

Figure 9. Trading Silicon Resources Between L2 Data Cache and Translation (Figure 10 is a zoomed in view of this graph normalized to the static, 1 cache tile case)

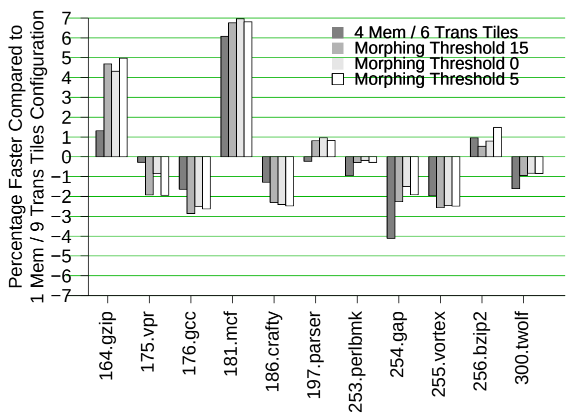

Figure 10. Relative Comparison of Performance for Differing Configurations (higher is better)

configurations. In this experiment we trade-off the number of translation tiles against the size of the L2 data cache. The first two designs are static configurations. The leftmost bar is the data for a configuration with 1 tile devoted to being a L2 data cache of 32KB. The next dataset is for a static configuration which utilizes 4 tiles as a L2 data cache, but has 3 fewer translators. Finally we have three configurations which dynamically change between the two previous static configurations. These implementations demonstrate dynamic reconfiguration in a parallel translator environment.

To better examine these results, Figure 10 contains the results normalized to the 1 cache tile configuration as a percentage faster or slower. The first thing to note is that the 4 tile L2 data cache configuration performs better than the 1 tile L2 cache configuration on some benchmarks and worse on others. This motivates static reconfiguration, or choosing the best virtual architecture on a per application basis. The larger cache configuration achieves superior performance on applications that have more demanding memory requirements. Next we turn our attention to dynamically reconfiguring the virtual architectures at runtime. As Figure 10 illustrates, on gzip, mcf, parser, and bzip2, when utilizing a dynamic reconfiguration system that introspectively examines the program and configuration, it is possible to beat the best static configuration. It is encouraging that even with reconfiguration occupancies, dynamic reconfiguration is able to reconfigure the virtual machine to best suit an application within a single execution.

Dynamic reconfiguration did not beat the static configurations on all the benchmarks. In these cases a virtual machine architect has the choice of using a static configuration per application that best suits that particular application. We investigated whether the reason that dynamic reconfiguration did not beat the static configurations on all applications was related to the reconfiguration heuristic. In this example, the reconfiguration heuristic was based off of the lengths of the “blocks to be translated” queues. We varied these metrics and found that as the threshold for reconfiguration was lowered from a length of 15 to a length of 0 (if anything was in a queue), that the number of reconfigurations increased. The overall performance though was not related to this, but largely decoupled from the reconfiguration heuristic.

Finally, while the gains due to dynamic reconfiguration over static configurations is humble in this implementation, we believe this is largely due to the parameters that are modified (L2 data cache size and translation resources) being second order factors in overall performance. With this implementation, dynamic reconfiguration is able to achieve a 3% performance increase over the best static configuration on some applications. This is quite encouraging and validates the idea that dynamic reconfiguration could lead to larger gains when applied to larger portions of a virtual architecture system such as the number of functional units.

4.5 Analysis of Performance Loss

The approach taken in this paper has focused on building a parallel dynamic translation environment built completely in software without any modifications to the underlying Raw hardware. By choosing this approach, the results presented in this section have been severely impacted by the mis-match in architectures and by some primitive facilities not being present in the Raw hardware such as memory translation and lack of a hardware instruction cache. In this section, we investigate where the performance is lost and suggest solutions to accelerate binary translation.

One of the primary differences between the Pentium III

Intrinsic	Raw Emulator	PIII
L1 Cache Hit	lat. 6, ocu. 4	lat. 3, ocu. 1
L2 Cache Hit	lat. 87, ocu. 87	lat. 7, ocu. 1
L2 Cache Miss	lat. 151, ocu. 87	lat. 79, ocu. 1
Exec. Units	1	3

Figure 11. Architecture Intrinsics

and our emulator is in the memory system. As Table 11 shows, the memory latency and occupancy for loads on the two architectures are quite different. Due to the lack of a hardware memory management unit on Raw, the emulator's load occupancy is 4 cycles for a L1 hit, while the occupancy on the Pentium III is 1 and has a lower latency. We can compute an estimate for how much performance is lost due to the memory system by using the memory system statistics for SpecInt gathered in [7]. We use basic CPI calculations as prescribed by $(memory\_access\_rate * (((1 - L1\_miss\_rate) * L1\_hit\_occupancy) + (L1\_miss\_rate * (((1 - L2\_miss\_rate) * L2\_hit\_occupancy) + (L2\_miss\_rate * L2\_miss\_occupancy))))) + ((1 - memory\_access\_rate) * non\_memory\_CPI)$ and hold the memory_access_rate and non_memory_CPI constant. Assuming a non_memory_CPI of 1, we compute a CPI of 3.9 based off of occupancy for the emulator and a CPI of 1 for the Pentium III. This assumes that both architectures can effectively find work to do to cover the latency of loads. With these assumptions, the emulator loses a factor of 3.9x when compared to the Pentium III due solely to memory system differences. The addition of a MMU to the Raw architecture would largely mitigate these differences. A MMU would primarily reduce the cost of an aligned L1 cache hit to one cycle of occupancy instead of 4.

The second significant factor where the emulator is inferior in performance relative to the Pentium III is in realized ILP. The Pentium III is an out-of-order three way superscalar. While the emulator presented in this paper does schedule instructions to hide functional unit latencies, the instructions still only execute on an in-order single-issue tile. We approximate the ILP inherent in SpecInt by looking to previous work done in [5]. The ILP for SpecInt 95 on a Pentium Pro was found to be 1.3. To a first order we approximate the ILP for SpecInt 2000 to be the same. Thus, the Pentium III has a 1.3x speedup when compared to our emulator. A portion of this ILP can be achieved in by our emulator if the Raw tile was multi-issue or if we focused on scheduling ILP across multiple tiles.

The x86 instruction set contains condition codes, while the Raw architecture does not. In order to emulate this architectural difference, our x86 emulator keeps the x86 flags packed in a register and uses insert and extract operations to access them. While our emulator does extensive dead flag elimination to reduce flag calculation, conditional branches require the use of a single flag that needs to be extracted from the packed flag register. This in effect turns any conditional branch from one instruction into two instructions. Assuming branches occur once every ten instruction, we can estimate the overhead cost of generating two instruction for every branch to be a 1.1x slowdown. One way to mitigate this cost is to add flags to the host architecture as Transmeta did. Another solution is to use a more sophisticated branch transformation optimization process that transforms compare and branch instructions to native host branch types.

If we account for the previously discussed fixable architectural deficiencies, we find that a slowdown of $3.9 * 1.3 * 1.1 = 5.5$ was minimally expected. This leaves only a factor of 1.3x of unaccounted slowdown for applications on the low-end of the slowdown spectrum (gzip, mcf, parser, bzip2). Of this 30% slowdown, we believe the following factors to be major factors, code translation cost, code caching overhead and non-optimal code generation.

While this analysis leaves only a factor of 30% of unaccounted slowdown on the low end of the slowdown spectrum, a 20x slowdown is unaccounted for in applications at the high end of the slowdown spectrum (gcc, crafty, vortex). In order to account for this disconnect, we refer back to Figure 6. As can be seen, the three applications with the most sever slowdown are also the applications with the greatest number of L2 code cache accesses. These poorly performing applications are approximately one hundred times more likely to access the L2 code cache per dynamic instruction than applications than perform well. This suggests that the instruction working set size for these applications is larger than the code cache contained on a single tile. Exacerbating this problem is the fact that the Raw host architecture does not have a hardware instruction cache, thus chaining is not possible outside of the lowest level of code cache. If the Raw host architecture were to add a hardware instruction cache, the lowest level code cache could be large enough to hold the instruction working set. By increasing the size of the lowest level of the code cache, chaining could be performed throughout the instruction working set. Also, an efficient hardware instruction cache would determine the most pertinent code to have in a tile's cache.

5 Future Work

In the future we believe that building virtual architectures that utilize dynamic translation technology will be an effective way to utilize the ever growing number of parallel resources on a single chip. To that end, we would like to extend our work to utilize as many processors as is possible. While we acknowledge that at some point there will be diminishing returns, if performance of single threaded legacy applications continues to be important, utilizing otherwise unused parallel resources to accelerate these legacy application can still provide much needed performance improvement.

To utilize larger arrays of on-chip processors, we are interested in extending this work to build more sophisticated virtual processors. There is potential to construct an out-of-order superscalar as a virtual architecture across an array of tiled processors. Sets of tiles can be dedicated to each of the functions that are typically employed in out-of-order superscalars such as register renaming, multiple functional units, instruction scheduling, and a reorder buffer.

Another way to improve the performance of our translator is to add hardware support. We are interested in generalized hardware that can aid in emulation of all architectures and not only x86. An example of hardware that we have considered adding to tiled processors is that of hardware to handle TLB lookups quickly. In translation systems, there exist two address spaces, the address space that the translator needs and an address space that the guest architecture utilizes. We think the addition of specialized loads and stores that have hardware TLB support for the guest architecture would be a large performance win. Also, the ability to cache miss to a differing tile instead of DRAM would be beneficial to employing virtual on-chip data caches built out of multiple tiles.

Very few of the findings of this work are specific to x86. Rather we chose x86 on Raw as a case study and because we felt that x86 was the most challenging guest architecture. An extension to this work is the support of multiple architectures such as PowerPC, SPARC, or Alpha all on the same tiled substrate and make a completely universal processor.

We would like to extend the dynamic reconfiguration ideas presented in this paper. We think that a fertile ground to investigate is how morphing can be applied to multiprocessor systems. We envision a large tiled fabric running many virtual x86's all at the same time. This would either be an x86 server farm or an x86 SMP all built virtually on a chip. If dynamic reconfiguration is then applied *between* virtual x86 processors, the virtual processors would compete for resources and this leads to a higher utilization of the underlying tiled fabric of processors. An example of this is if two virtual x86's share 16 tiles. If one of the x86 processors is stalled waiting on I/O while the other is crunching numbers, the stalled processor could be shrunk down to one tile while the computationally bound x86 could use the remaining 15 tiles to speed up its execution.

Lastly, we would like to improve on this work in both performance and robustness. We believe that there is significant work that can go into improving the code quality post translation and hence performance wins. Also, finding and mapping ILP onto multiple functional units would improve performance. On the robustness front, we would like to turn our current system into a full system emulator. Currently we support only userland codes with a proxy system call interface. To support booting a complete operating system such as Windows or Linux, system level instructions need to be added to the core translator. Also, we would like to be able to support self modifying code and 16-bit addressed code. The current emulator was designed with self modifying code in mind and is currently capable of detecting writes to memory pages which contain code that has been translated. Ultimately we are striving to run arbitrary x86 operating systems and applications such as Windows and Microsoft Office.

6 Related Work

The work presented in this paper builds on the work previously done in the dynamic translation, optimization, and recompilation communities. Much of the early work in this field, such as how to manage code caches and the introduction of chaining, was done in Shade [8] and Embra [24]. In our system we apply optimizations during translation of the code. There exist several dynamic code optimizers such as Dynamo [2], DynamoRio [6], rePLay [18], and numerous other projects that optimize code for JVMs. This work extends previous dynamic translation work by recasting many of these ideas in a parallel, spatially aware, environment.

DAISY [11], DELI [10], and Transmeta [9] all investigate dynamic translation to a VLIW architecture. The work presented in this paper differs from these previous parallel mappings, by applying parallelism in a much more coarse grained manner. Our work investigates building complete virtual architectures by using a tiled processor as a computational fabric that is mapped onto much in the same way a FPGA is used. These previous projects focused more heavily on mapping translated code across a VLIW to find ILP. We believe this work is complementary to our work, and we hope to utilize some of their parallel mapping methods in the future. Another difference between our work and these previous attempts is that mapping ILP across a tiled architecture can be more complicated but provides more peak parallelism than is available in most typical VLIWs.

This work is motivated by new emerging parallel architectures on a chip. Previously built chip multiprocessors include Piranha [4], the POWER 4 [22], and MAJC [23]. Current tiled and replicated processor projects include Trips [16], Wavescalar [19], Smart Memories [14], and Raw [20]. Recently we have seen the adoption of multicore design ideas from major industry processor companies. We hope that the work in this paper motivates further design of these parallel architectures, and could possibly even motivate the addition of hardware to better support parallel emulation.

One approach to supporting legacy ISAs is to integrate customized hardware to execute them. Early Itanium processors take this approach by integrating hardware x86 support [17]. Unfortunately integrating specialized hardware precludes using this “legacy” silicon area for execution of recompiled parallel applications. Also, adding specialized x86 hardware does not accelerate the execution of other legacy ISAs such as PowerPC or multiple x86's.

7 Conclusion

In this work, we investigated using a parallel dynamic binary translation engine to exploit the parallel resources available in on-chip multiprocessors such as CMPs, tiled processors, and multi-core processors to accelerate the execution of legacy programs on these novel architectures. We introduced three mechanisms that can be applied to accelerate emulation in a parallel environment and demonstrated their effectiveness in a real-world working proof of concept prototype system that runs x86 binaries on the Raw tiled processor. We hope that this work is able to guide future on-chip parallel processor designs and provide them a mechanism for efficiently executing legacy applications.

Acknowledgments

Thanks to Walter Lee for the automated Makefile infrastructure to gather SpecInt results. This work is funded by DARPA, the NSF, and MIT's Project Oxygen.

References

[1] Press release: Intel silicon innovation to shape direction of the digital world: Multi-core processors, other key silicon technologies part of platform approach, Sept. 2004.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[3] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[4] L. A. Barroso et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 282–293, June 2000.

[5] D. Bhandarkar and J. Ding. Performance characterization of the Pentium Pro processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 288–297, Feb. 1997.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 265–275, Mar. 2003.

[7] J. F. Cantin and M. D. Hill. Cache performance for SPEC CPU2000 benchmarks: Version 3.0, 2003. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[9] J. C. Dehnert et al. The Transmetta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[10] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *Proceedings of the International Symposium on Microarchitecture*, 2002.

[11] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the International Symposium on Computer Architecture*, pages 26–37, June 1997.

[12] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3):54–59, Mar. 2000.

[13] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[14] K. Mai et al. Smart Memories: A modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 161–171, June 2000.

[15] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[16] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 422–433, June 2003.

[17] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept. 2000.

[18] B. Slechta et al. Dynamic optimization of micro-operations. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 165–176, Feb. 2003.

[19] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the International Symposium on Microarchitecture*, pages 291–302, Dec. 2003.

[20] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar. 2002.

[21] M. B. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[22] J. Tendler, J. Dodson, J. J.S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002.

[23] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, Nov. 2000.

[24] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.