Dynamic Optimization of Micro-Operations

Brian Slechta David Crowe Brian Fahs Michael Fertig Gregory Muthler
Justin Quek Francesco Spadini Sanjay J. Patel Steven S. Lumetta

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

Inherent within complex instruction set architectures such as x86 are inefficiencies that do not exist in a simpler ISAs. Modern x86 implementations decode instructions into one or more micro-operations in order to deal with the complexity of the ISA. Since these micro-operations are not visible to the compiler, the stream of micro-operations can contain redundancies even in statically optimized x86 code. Within a processor implementation, however, barriers at the ISA level do not apply, and these redundancies can be removed by optimizing the micro-operation stream.

In this paper, we explore the opportunities to optimize code at the micro-operation granularity. We execute these micro-operation optimizations using the rePLay Framework as a microarchitectural substrate. Using a simple set of seven optimizations, including two that aggressively and speculatively attempt to remove redundant load instructions, we examine the effects of dynamic optimization of micro-operations using a trace-driven simulation environment.

Simulation reveals that across a sampling of SPECint 2000 and real x86 applications, rePLay is able to reduce micro-operation count by 21% and, in particular, load micro-operation count by 22%. These reductions correspond to a boost in observed instruction-level parallelism on an 8-wide optimizing rePLay processor by 17% over a non-optimizing configuration.

1 Introduction

Complex instruction set architectures present significant design challenges for high-performance implementations. Variable-length instruction formats and high-granularity instructions complicate the decoding and execution processes. In order to deal with ISA complexities, current implementations decode instructions into simpler micro-operations. These micro-operations are essentially control words that traverse the processor pipeline and perform an equivalent amount of processing to the instructions of a simple ISA.

As complex instructions are individually decoded into constituent micro-operations, an opportunity is created to globally optimize their constituent micro-operations. For example, a basic block's micro-operations, optimized as a unit, can be more efficient than micro-operations generated one instruction at a time. For simpler ISAs, such as PowerPC, SPARC, and MIPS, this type of optimization is performed during compilation, as the emitted code is effectively at the same level as the micro-operations of a complex ISA. This optimization opportunity thus exists with complex ISAs, even when basic blocks are statically optimized. Leveraging this opportunity, of course, requires raising the architectural state boundaries to the basic block level. That is, architectural state is only guaranteed to correspond at basic block boundaries.

The x86 instruction set architecture in particular can benefit from optimizations at the micro-operation level. The x86 ISA provides only a few 32-bit registers for storing temporary values, which constrains optimization at compile time. The ISA's two-address instruction format, along with the high-granularity of some common instructions, also increase the opportunity for micro-operation optimization. Furthermore, non-uniform instruction semantics, such as opcodes that requires specific source registers (e.g., the x86 DIV instruction), limit a compiler's ability to efficiently allocate registers. Optimization at the micro-operation level partially alleviates these problems because the register assignment is not restricted to the architectural register space.

In this paper, we evaluate the potential of performing optimizations on regions of x86 micro-operations, where the regions are larger than basic blocks. Using the rePLay Framework as our microarchitectural substrate, we evaluate a processor architecture that performs micro-operation optimization on atomic dynamic instruction traces, or *frames*. rePLay contains hardware support for performing optimizations (via an optimization engine) and hardware support for speculation recovery (enabling speculative optimization). The optimizer performs simple optimizations on each frame, such as dead code elimination, reassociation, and store forwarding. We study the effect of optimization using trace-driven simulation on both SPECint 2000 benchmarks and commercial x86 applications. On average across this spectrum of applications, we observe that optimization with rePLay reduces micro-operation count by 21% and, in particular, load micro-operation count by 22%. These reductions boost the instruction-level parallelism of a deeply-pipelined 8-wide optimizing rePLay processor by 17% over a non-optimizing configuration across the applications.

In this paper, we provide two major contributions over the previous work on rePLay [4, 13]. First, we evaluate the impact of micro-operation-level optimization in the context of the x86 ISA. Previous work examined hardware-based dynamic optimization of Alpha instructions. This distinction is important: the granularity of the x86 instruction set and its limited register set create inefficiencies that are more prevalent than in binaries of simpler ISAs. The full benefit of x86 optimization requires dealing with memory dependencies, and we describe our scheme for speculatively optimizing around them. We present our evaluation on continuous traces (i.e., including DLL calls and system code) of SPECint benchmarks and real applications.

Second, we describe the high-level microarchitecture of a programmable optimization engine datapath that can be used to perform micro-operation optimizations. The complexity of the optimization datapath and the optimization software is reduced by utilizing three properties of frames: (1) they are atomic, (2) they embody a single control path, and (3) the register renaming process renders the frames into a form amenable to optimizations by guaranteeing that each operation writes to a unique physical register.

The remainder of the paper is organized as follows. The next section provides an overview of the rePLay microarchitectural substrate. Section 3 contains examples of the micro-operation optimizations that we evaluate in this paper. Section 4 introduces the primitive operations and structure of the optimization datapath. The experimental infrastructure is described in Section 5, and experimental results are presented in Section 6. Section 7 contains related work. Section 8 provides conclusions.

2 The rePLay Microarchitecture

In this section we provide an overview of the rePLay Framework, which is designed to facilitate aggressive dynamic optimization with low overhead. To this end, the framework consists of five key components: (1) a frame constructor for creating candidate optimization regions, (2) a programmable engine for optimizing these regions, (3) a frame cache for storing these regions on-chip, (4) a component for sequencing between regions, and (5) a mechanism to recover architectural state when speculative optimizations prove incorrect. These components are integrated into a processor's fetch and execution engine, as shown in Figure 1.

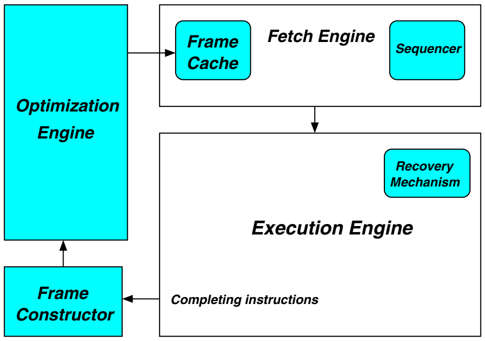

Figure 1. The rePLay mechanism coupled to a processor architecture.

A central concept of rePLay is the concept of the atomic region, or *frame*. A frame is similar to a *trace* in a trace-scheduling compiler [5] or a *block* in the Block-Structured ISA [8, 11]. All control dependencies within a frame are removed, ensuring that all instructions within the frame are mutually control independent. In particular, either all instructions in a frame commit their results, or none of them do. This atomicity simplifies the optimization algorithms used and allows high-bandwidth instruction fetch. To remove control flow, rePLay's frame constructor converts dynamically biased branches into assertions, then merges sequences of constituent basic blocks into frames. Using frames as an optimization entity is of particular importance because precise architectural state need only be maintained at frame boundaries, thus allowing the optimizer more leeway in performing optimizations. We demonstrate this in Section 3 with an example.

An assertion whose condition is not true triggers a hardware recovery event that rolls back architectural state to the beginning of the frame. A rollback mechanism of this type is already present in most modern processors to support out-of-order, speculative execution. This restoration capability implies that any state generated during frame execution must be buffered until all assertions within the frame have executed successfully (not fired). Once all assertions within a frame have been checked and all micro-operations have been executed, state changes generated within the frame can be committed. This atomicity enables the optimizer to make speculative optimizations safely, with the speculative assumptions enforced by assertions.

With this general design, frame construction can be done in hardware or by the compiler as a specification of the ISA. All of our current work has focused on hardware-level frame construction, but this property is not inherent to rePLay.

3 Micro-operation Optimization

Processors that implement complex ISAs like x86 typically decode each instruction into simplified, fixed-format micro-operations. As instructions are decoded independently

; instructions	**unoptimized micro-operations**	**intra-block optimization**	**inter-block optimization**	**frame-level optimization**
PUSH EBP	01 SS:[ESP - 04H] ← EBP	01 SS:[ESP - 04H] ← EBP	01 SS:[ESP - 04H] ← EBP	01 SS:[ESP - 04H] ← EBP
	02 ESP ← ESP - 04H	; incorporated into 04′		
PUSH EBX	03 SS:[ESP - 04H] ← EBX	03′ SS:[ESP - 08H] ← EBX	03 SS:[ESP - 08H] ← EBX	03 SS:[ESP - 08H] ← EBX
	04 ESP ← ESP - 04H	04′ ESP ← ESP - 08H	04 ESP ← ESP - 08H	; incorporated into 16′
MOV ECX,[ESP+0CH]	05 ECX ← [ESP + 0CH]	05 ECX ← [ESP + 0CH]	05 ECX ← [ESP + 0CH]	05′ ECX ← [ESP + 04H]
MOV EBX,[ESP+10H]	06 EBX ← [ESP + 10H]	06 EBX ← [ESP + 10H]	06 EBX ← [ESP + 10H]	06′ ET0 ← [ESP + 08H]
XOR EAX,EAX	07 EAX,flags ← 0	07 EAX,flags ← 0	07 EAX,flags ← 0	07 EAX,flags ← 0
MOV EDX,ECX	08 EDX ← ECX	; reassociated, then removed		
OR EDX,EBX	09 EDX,flags ← EDX | EBX	09′ EDX,flags ← ECX | EBX	09 EDX,flags ← ECX | EBX	09′ EDX,flags ← ECX | ET0
JZ 15H <Block2>	10 **if** (flags = 0) **jump** Block2	10 **if** (flags = 0) **jump** Block2	10′ **if** (flags ≠ 0) **exit**	10′ **assert** flags = 0
; jump is typically taken				
Block2:				
POP EBX	11 ESP ← ESP + 04H	; incorporated into 16′		
	12 EBX ← SS:[ESP - 04H]	12′ EBX ← SS:[ESP]	12 EBX ← SS:[ESP]	; forwarded from 03
POP EBP	13 ESP ← ESP + 04H	; incorporated into 16′		
	14 EBP ← SS:[ESP - 04H]	14′ EBP ← SS:[ESP + 04H]	; forwarded from 01	
RET	15 ET2 ← SS:[ESP]	15′ ET2 ← SS:[ESP + 08H]	15 ET2 ← SS:[ESP + 08H]	15′ ET2 ← SS:[ESP]
	16 ESP ← ESP + 04H	16′ ESP ← ESP + 0CH	16 ESP ← ESP + 0CH	16′ ESP ← ESP + 04H
	17 **jump** (ET2)	17 **jump** (ET2)	17 **jump** (ET2)	17 **jump** (ET2)

Figure 2. Impact of optimization scope on a procedure from crafty. The prime notation denotes changes to sources and/or results relative to the previous level of optimization.

from one another, the resulting micro-operation stream can contain inefficiencies. For example, an instruction's micro-operations might calculate an intermediate value already calculated in a previous instruction's micro-operations. The objective of this paper is to identify and remove such inefficiencies automatically, and to quantify the resulting benefit to performance. To illustrate the optimization opportunities more clearly, we now examine the impact of optimizations based on the scope in which they are applied, using a code example.

The benefit of optimizing micro-operations depends strongly on the scope of the optimizations. Simple local optimizations, such as elimination of NOPs, can be performed before filling into a trace cache. The granularity of x86 instructions prevents a compiler from fully optimizing code, leaving room for *intra-block* optimizations across instructions within a basic block. Considering several blocks creates *inter-block* opportunities, as control enters only at the first block, ensuring that early blocks have been executed before later blocks. Finally, additional optimizations become feasible when the possibility of early exit is eliminated by treating a sequence of blocks atomically. In this case, *frame-level* optimizations can assume that subsequent blocks are always executed.

In this section, we describe the optimizations performed by the rePLay optimization engine and illustrate their impact as a function of scope through the use of a running example, shown in Figure 2. In Section 4, we describe the hardware primitives necessary to support these optimizations and describe the structure of the rePLay optimizer in more detail. The two basic blocks in the running example form part of a larger frame from crafty, a chess-playing code from SPECint 2000. The first column shows x86 instructions, and the second column shows micro-operations. The remaining columns illustrate the benefits of optimization on micro-operations.

3.1 Intra-block optimization

The third column of Figure 2 illustrates optimization within a basic block. Within the first block, the stack updates due to the two PUSH instructions can be merged into a single update. Recall that all optimizations are performed using renamed registers. First, *reassociation* replaces the uses of the result of micro-operation 02 with uses of the live-in ESP register. The offset in 03′ and the immediate value in 04′ incorporate the subtraction performed by 02. As the result produced by 02 is no longer used, and is not a live-out value of the block, *dead code elimination* removes the micro-operation. The same optimizations eliminate micro-operation 08, which the compiler uses to avoid a second load of the value in ECX along the non-taken path following 10. A three-operand instruction (OR EDX,ECX,EBX) serves the same purpose, but none is available in the x86 ISA. The micro-operation format does support three operands, and the two micro-operations are combined into 09′ by the optimizations. The second basic block also allows intra-block optimization of stack manipulations.

3.2 Inter-block optimization

The fourth column in Figure 2 illustrates optimization when a single entry point is assumed, as is typically the case in a trace cache. Multiple exits—after the first basic block in the figure, for example—are still possible. *Store forwarding* associates the load in micro-operation 14 with the store in 01. As the result of 14 is live-out in EBP, and the value stored by 01 is live-in in the same register, the load is eliminated. In contrast, store forwarding does not occur for EBX, which is modified by the first block and must be correct should control exit at micro-operation 10′. The need to maintain all live-out registers from the first block thus limits inter-block optimization. Some inter-block benefits can be obtained with a compiler through techniques such as tail duplication, but instruction sets like the x86 make these techniques less effective.

3.3 Frame-level optimization

The concept of an atomic frame restricts the model commonly used in a trace cache by requiring a unique exit point. This restriction eliminates the need to maintain proper architectural live-out values for intermediate exits. Effectively, the entire frame can be optimized as a basic block.

The rightmost column in the example shows optimization of the blocks as a rePLay frame. Micro-operation `10'` is changed to an assertion, which allows no exit. Stack updates in the two blocks are merged into a single update. Store forwarding eliminates `12`, and temporary register ET0 is substituted for the intervening use of register EBX in `06'` and `09'`. Overall, seven of the seventeen micro-operations are removed, including two of the five loads.

When this fragment is optimized in the context of a larger frame, the results are even more impressive. Typically, code from the procedure's call site precedes and follows the blocks shown, allowing the parameter loads in `05` and `06` to be removed. The load of the return address in micro-operation `15` is also eliminated, and the stack update in `16` is folded into an update outside of the procedure. Finally, *constant propagation* from the call site identifies the return jump in `17` as a constant target and removes it, as the target block is included in the frame. In some cases, constant propagation can also eliminate micro-operation `07`, which places the procedure's return value into EAX. The optimizations thus reduce the entire procedure to two stores and a single check (`09` and `10`) consisting of an ALU operation and an assertion. While not all code sequences are so amenable to optimization, the example highlights the degree to which code generated for complex ISAs can be optimized by dynamic hardware mechanisms such as rePLay.

3.4 Additional optimizations

Several rePLay optimizations were not mentioned in the context of our running example. *Common subexpression elimination* serves primarily to remove redundant loads, which often appear when x86 loops are unrolled within a frame. *Value assertion optimization* combines the typical x86 sequence of a flag-generating instruction such as CMP (compare two operands) followed by a conditional branch into a single micro-operation.

Store forwarding and redundant load elimination are also allowed to occur speculatively within frames. Consider the case in which a load follows a store to the same address (same register and offset), but is separated by one or more stores with different address registers. Only when these stores do not alias to the store-load pair can the stored value be forwarded to the load. We record aliasing events during execution and pass this information to the optimizer. If the intervening stores did not alias during execution, the optimizer speculates that they never alias, and removes the load. The intervening stores are marked as *unsafe stores*. A similar optimization is performed when redundant loads are separated by stores that may alias.

When an unsafe store executes, its address is compared against all other memory transactions (both loads and stores) prior to it in the frame. If an unsafe store conflicts with any other transaction, the frame is aborted and the original instructions are executed instead. No optimization removes stores, thus all unsafe stores execute when a frame is fetched. In practice, loads removed speculatively almost never cause frames to abort, but represent a substantial fraction of loads in a dynamic micro-operation stream, as discussed in Section 6.

4 Optimizer Design

In this section, we provide some details on the design of the rePLay optimizer datapath. Most prior work on optimizer design has focused on identifying the potential of hardware-based optimizations. Little has been done on specifying the design of a hardware optimizer or its interface to the optimization software. Here, we provide a high-level description of the types of hardware primitives that an optimizer should support in order to facilitate low-latency optimization.

In our previous work [4], we demonstrated that a pipelined frame optimizer with a latency of 1K-10K cycles could reasonably match the latency and throughput requirements for a rePLay system.

In order to achieve such low latency, the optimizer hardware must provide useful primitives that are central to a wide range of code optimization algorithms. In particular, the hardware optimizer should provide three classes of primitives: (1) It should provide the ability to quickly retrieve one of an instruction's parent instructions (we use the term *instruction* in this section as a more general form of *micro-operation*), and to retrieve an instruction's children. That is, the optimizer should support quick traversal of a frame's dataflow graph. (2) The hardware needs to support general field extraction and bit manipulation operations, for example to isolate an instruction's opcode field. Such primitives are useful for testing and modifying various bits within instructions. (3) The hardware should support the ability to add and remove instructions in a frame.

To facilitate the optimizer design, we first render frames into a form where each operation writes to a renamed register. That is, a physical destination is written only once within a frame. No write-after-write or write-after-read register naming conflicts exist in a frame. In this form, an instruction's physical source register numbers uniquely identify the instruction's parents. It should be noted that a frame whose instruction register operands have been renamed by the processor is already in this form, except for the occasional reuse of a physical register within a frame.

Figure 3 shows the major components of the optimizer.

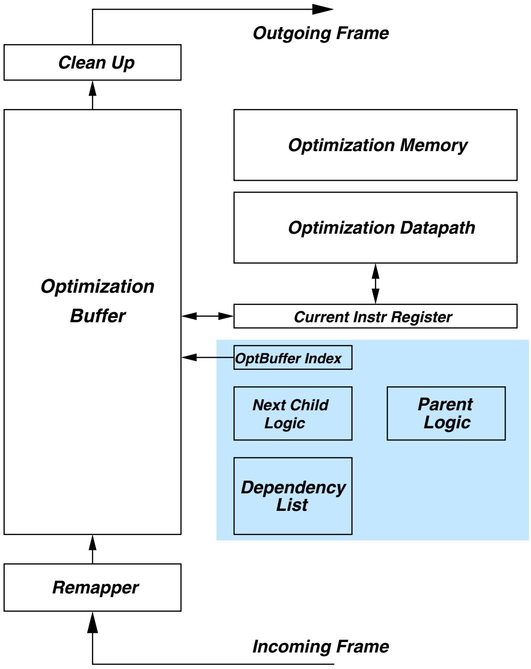

Figure 3. The rePLay Optimizer.

An unoptimized frame is added to the optimizer's optimization buffer. Each instruction in the frame has its destination register remapped such that an instruction placed in location **m** of the buffer will write physical register **m**. With this property, given an instruction, retrieving the parent instruction that produced its SourceA, for example, is a trivial lookup. This remapping also alleviates any reuses of physical registers. Depending on the physical register allocation strategy of the processor, this Remapping can be a modification of the physical register assignment done by the Register Alias Table (RAT) prior to execution. In the worst case, the Remapping process is the same as the register renaming process, except that it need not operate at the same high bandwidth. A remapping bandwidth of one or two instructions per cycle is likely to be sufficient.

Once Remapped, a frame resides in the optimization buffer until the optimization process is complete. If a frame arrives while another is being optimized, the arriving frame must be buffered or dropped. Alternatively, the optimizer can be pipelined to permit optimization of frames concurrently. (We model a variable optimization latency of 10 cycles per instruction in a pipelined optimizer. Simulation results show that a pipeline depth of 3 is sufficient to sustain the throughout of our rePLay model.) The OptBufferIndex is used to select an instruction to read out of the buffer. Due to the Remapping policy, reading the buffer at index **m** provides the micro-operation that generates physical register **m**.

The format of the optimization instruction (or in this case, micro-operation) is provided in Figure 4. The Remapping process provides each instruction with new source physical registers and a new destination. Instructions that produce a live-out value, or use a live-in, are marked explicitly. This explicit renaming of internal values within a frame can facilitate high-speed instruction renaming when the frame is fetched [15, 16]. When the frame is fetched, only the instructions that require live-in values need to read the RAT, and only those instructions that produce live-outs need to update the RAT. Physical register assignment for values internal to the frame can then be done without a table lookup.

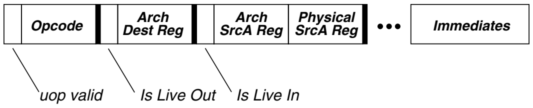

Figure 4. The optimizer's micro-operation format.

Many optimization algorithms require traversing though a dataflow graph in order to identify optimization opportunities. To support this process without requiring the software component of the optimizer to build and maintain the dataflow flow graph for every new frame, the optimizer hardware supports traversal via the logic in the shaded region of the Figure 3. The Parent Logic and Next Child Logic enable the optimizer to move from an instruction to its parents or children. Moving from an instruction to the parent that produced its SourceB input, for example, is trivial. The SourceB physical source register number is the producer's index. Moving from an instruction to its children is trickier, as an instruction can have many children. This operation requires maintaining a hardware Dependency List structure that associates an instruction with its children. This structure enables the Next Child Logic to iterate over the list of children, enabling optimizations that require visiting all children of an operation. Also, the traversal logic can increment and decrement the OptBufferIndex to support simple iterations through a frame's micro-operations.

Bit manipulations and field extractions are provided by the simple optimization datapath, which contains an ALU and a load/store port to the optimization memory. This hardware is useful for recalculating immediate values for the reassociation optimization, or for comparing base registers and offsets for memory optimizations such as store forwarding. The software portion of the optimizer itself resides in the optimization memory and is activated by the arrival of a new frame. In Figure 3, we show only one current instruction register for simplicity. Implementations may have several instruction registers that are directly accessible by the datapath, for example to access both a micro-operation and one of its parents simultaneously.

The third set of primitive operations is that of adding/removing instructions from the optimization buffer. To remove an instruction, the instruction is simply marked invalid and the instruction is removed from its parents' lists in the Dependency List structure. Adding an instruction is more involved (and less frequently needed): the new instruction is written into a spare instruction slot at the end of the frame, and the parents' dependency lists are updated. By construction, the instructions of a frame are explicitly in renamed form and can be arbitrarily reordered—an instruction can appear before its parents. But this end-of-frame insertion cannot be done when memory operations are involved because memory ordering must be preserved. The optimizer is therefore prohibited from inserting new loads and stores (in particular those micro-operations that affect memory ordering).

Rescheduling or repositioning code is accomplished in the final stage using the Cleanup Logic. During the optimization process, a position field encoded with the micro-operation is used to identify the micro-operation's final position in the completed frame. By default, the frame remains in the order it appears in the buffer. However the optimization algorithms can use the position field to adjust the frame's schedule. The Cleanup Logic can use associative lookups to read the frame out of the Optimization Buffer in the specified order. At this point, any invalidated instructions are deleted from the frame.

5 Experimental Setup

In this section, we describe the experimental setup used to evaluate our mechanism for dynamic optimization of micro-operations. First, we describe the trace-driven simulation environment. Second, we describe the experimental workload set. Finally, we present and motivate our baseline processor architecture.

5.1 Simulation Environment

Our simulation environment is trace-driven with limited wrong-path support (wrong-path behavior is only modeled on asserting frames). The environment consists of four components: the Micro-operation Injector, Timing Model, State Verifier, and the rePLay Engine, as shown in Figure 5.

5.1.1 Micro-Op Injector

The Micro-Op Injector consists of a trace reader, which reads x86 trace files, and an x86-to-rePLay micro-operation translator.

The trace reader reads and disassembles the raw instruction data from a hardware-generated trace file. The trace files were generated on a Windows NT-based platform, and were provided by Advanced Micro Devices. Each trace record contains instruction data, register state changes, memory transactions, and interrupt information for each x86 instruction throughout a span of execution of an application. Each trace represents a particular "hot spot." That is, it is the dynamic trace of a code segment that ultimately accounts for a large amount of execution time.

The second stage of the Micro-Op Injector is the translator. The role of the translator is to convert the disassembled x86 instructions into our processor's native micro-operation format. We refer to this internal format as the rePLay ISA.

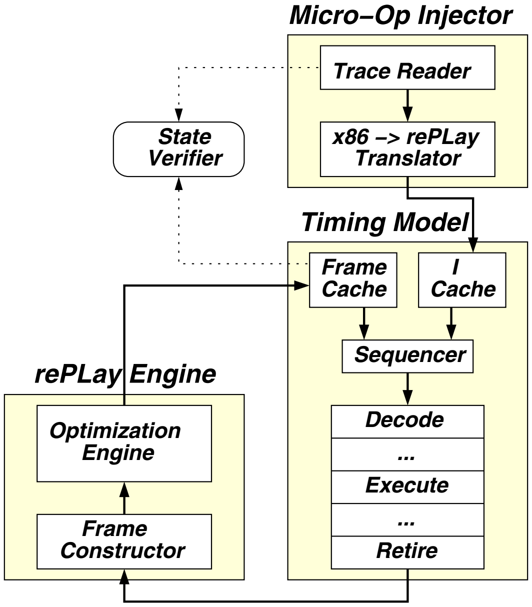

Figure 5. Simulation Environment

Since x86 micro-operations and x86-to-micro-operation decode flows of real implementations are kept proprietary, we modeled our ISA to be close to a generic RISC ISA. Our decode flows are fairly efficient, and in the end we attain an average micro-operation-to-x86 instruction ratio of 1.4, which is close to our estimates of what is achieved on real x86 implementations.

While the use of trace-driven simulation can be criticized for its lack of wrong-path effects, our traces allow us to include the effects of interrupts, system calls, and dynamically-linked libraries. We use the information contained in the trace to validate our execution environment and to handle very rare x86 instructions that might otherwise require substantial effort to implement (for example, modifiers of segment descriptors, call gates, etc. On such instructions, we flush the processor pipeline). Such long-flow instructions account for a very small portion of the dynamic instruction stream (less than 0.05% on the traces we evaluate).

5.1.2 Timing Model

The Timing Model contains models of the caches, memory system, branch predictors, and pipeline, and is used to determine the number of cycles needed to execute the sequence of continuous micro-operations provided by the Injector. When simulating a standard processor (without rePLay features), the micro-operation stream comes directly from the Injector. With rePLay features enabled, the micro-operations (possibly in optimized form) for a particular sequence of x86 instructions can also come directly from the frame cache.

The configuration of the timing model is parameterized, and for the experiments provided in this paper, we have chosen an 8-wide pipeline whose specific configuration is described in Section 5.3

5.1.3 State Verifier

In order to verify that the optimizations that are performed on the micro-operation stream are valid, we employ a state verifier. This checker is used to ensure that the state transformations (architectural register state and memory) made by an optimized frame are equivalent to those of the actual, unmodified instruction stream. The checker is valuable for testing the validity of our micro-operation decoder and optimizer.

In order to properly verify the execution of our micro-operations, the state verifier's job is two-fold. First, each x86 instruction that is emitted from the trace carries register state changes and any memory transactions associated with that instruction. Additionally, there is a memory address and a memory data field associated with each load and store. The load data is used by the verifier to perform the load operations while the store data is used solely to verify the stores that are performed by our micro-operations. After each x86 instruction is decoded and executed, the resulting state is compared to the trace.

The second role of the state verifier is to validate the functionality of the optimizer. This role is essential in preventing flaws in the optimization algorithms. Each frame is executed using the architectural state and memory state when the frame is fetched. The memory state consists of two maps, an initial memory map and a final memory map. Both maps are generated by first executing the corresponding original instructions from the trace. Since all loads within a frame are a subset of those loads that would be performed by the original instruction stream, we commit to the initial map the first load and store transactions from each live memory location in the trace. All store transactions in the trace are committed to the final map which is used to compare the memory state at the frame boundary. The frame is considered valid only if execution satisfies these requirements: (1) all loads can be found in the initial memory map, (2) all memory state affected by the trace is equivalently affected by the frame at the frame boundary, and (3) all architectural register state is equivalent at the frame boundary.

5.1.4 rePLay Engine

The last major component in the simulation environment is the rePLay Engine. Retired instructions from the processor pipeline flow into the frame constructor, which dynamically synthesizes long regions of code into atomic frames. Optimizations are then applied to these frames by the optimization engine before they are deposited into the frame cache. We model the optimization engine abstractly in this study—each frame is optimized with a variable latency of 10 cycles per instruction, and the optimizer is pipelined. Simulations show that a pipeline depth of 3 is sufficient to obtain the results we report.

5.2 Benchmarks

Our selection of benchmarks was driven primarily by availability. AMD graciously provided a set of traces, from which we selected the application set listed in Table 1. The set consists of 7 SPECint 2000 benchmarks and 7 desktop applications (all of which were executed as part of the Winstone benchmark suite). Some of the applications we examine actually consist of multiple trace files, each representing a different portion of execution of the application. The table above indicates the number of x86 instructions and the number of different traces a particular test application constitutes.

Name	Type of App.	Total x86 Insts.	Number of Traces
bzip2	SPECint	50M	1
gzip	SPECint	50M	1
crafty	SPECint	50M	1
eon	SPECint	50M	1
parser	SPECint	50M	1
twolf	SPECint	50M	1
vortex	SPECint	50M	1
Access	Business	200M	2
DreamWeaver	Content	200M	2
Excel	Business	300M	3
LotusNotes	Business	200M	2
PhotoShop	Content	200M	2
PowerPoint	Business	300M	3
SoundForge	Content	300M	3

Table 1. Experimental Workload

5.3 Processor Configurations

We evaluate several configurations based on a common processor pipeline. The pipeline is an 8-wide fetch, issue, retire pipeline (the width refers to micro-operations). The pipeline models 15 cycles between the fetch of a branch and the earliest possible point of its execution. Because the pipeline is deep, we model a speculative wakeup/scheduling that occasionally requires instructions to be rescheduled if a result is not ready when expected (applies primarily to dependent operations on data cache misses). The scheduling window is 512 micro-operations. The various properties of the processor appear in Table 2.

The major experiments that we report in the following section are based on a rePLay processor configuration consisting of a frame cache that can store 16k micro-operations (approximately equivalent to a 64kB ICache) and a 8kB ICache. When fetching from the ICache, the maximum rate through the x86 decoder is four x86 instructions per cycle. The rePLay frame constructor creates frames between 8 and 256 original micro-operations.

In some experiments we also compare to a Trace Cache configuration (16k micro-op Trace Cache/8kB ICache) in which the fill unit continuously creates traces with up to three branch micro-operations. We also provide, as a reference, a 64kB ICache configuration. In addition, in both the Trace-Cache and rePLay configurations, we model an idle cycle

Pipeline	8-wide fetch/issue/retire x86 decoders: 4 per cycle 15 cycles (min) for BR resolution
Predictor	18-bit gshare
Inst Window	512 instructions
ExeUnits	6 simple ALU 2 complex ALU 3 FPUs 4 load/store units
Frame/Trace Cache	16k micro-operations (approximately 64kB)
L1 DCache	32kB, 2 cycle hit 4 read and 4 write ports
L2 Cache	512kB, 10 cycle hit
Memory	50 cycles

Table 2. Configuration of Processor.

when switching between caches. We denote these cycles as Wait cycles.

6 Performance Evaluation

In this section, we evaluate the performance of the processor configurations described in the previous section. In particular, we investigate the impact of dynamic optimizations using the rePLay framework.

6.1 Basic evaluation

Figure 6 contains a simulation-based performance estimation of four configurations: ICache (**IC**), Trace Cache (**TC**), basic rePLay (**RP**), and rePLay with Optimization (**RPO**) all operating on a deeply pipelined 8-wide processor. The data is plotted in terms of average x86 instructions retired by the processor per cycle (IPC). For the RPO configuration, we plot the effective IPC which takes into account the number of original x86 instructions retired by the processor.

For all but one application, gzip, the optimizing rePLay configuration outperforms all others. The percentage increase in IPC over a non-optimizing configuration (RP) induced by the optimizer is shown on the graph near the individual bars representing RPO. On average, there is a 17% increase in IPC due to optimizations, but this increase is highly variable from application to application.

Even though the SPEC benchmarks were aggressively profiled and compiled with the Intel Proton Compiler, the average increase in parallelism across SPEC benchmarks is slightly higher than in the desktop applications. Despite aggressive static compilation, opportunity still exists for optimizing the micro-operation stream. This difference in parallelism is largely due to the difference in the dynamic coverage of frames over the original micro-operation stream from these benchmark suites. SPEC benchmarks exhibit about 86% coverage while the desktop applications only show about 72% coverage. Higher coverage leads to a greater opportunity for dynamic optimizations.

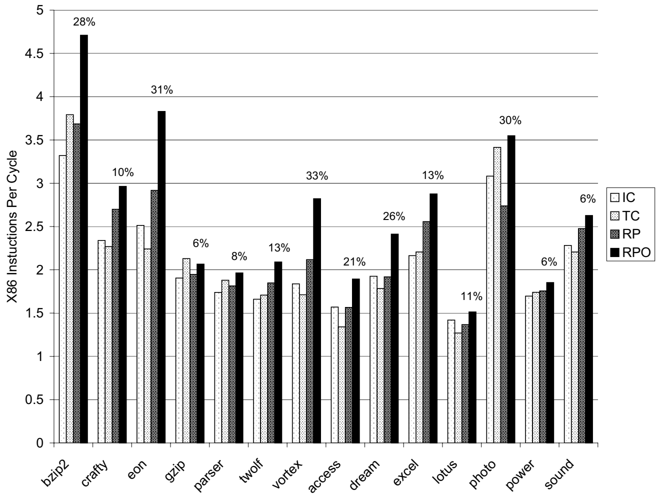

Figure 6. Estimated x86 Instructions Retire Per Cycle for the ICache, Trace Cache, rePLay, and rePLay+Optimization processor configurations. All configurations were evaluated in the context of an deeply pipelined 8-wide processor with 15 cycles (min) for branch resolution.

The rePLay optimizations offer two main sources of benefit: they reduce micro-operation count, and they reduce computation tree height. In subsequent experiments in this section, we try to isolate some of the individual effects that enable the optimizations to be effective.

Figures 7 and 8 show a cycle-by-cycle breakdown of the RP and RPO configurations. The breakdown is done from the perspective of the instruction fetch stage: if a cycle is spent fetching from the Icache, the cycle is tallied as an **ICache** cycle. If the fetch comes from the Frame Cache, it is a **Frame** cycle. Each cycle is uniquely categorized into one of seven bins, in the following order of priority: **Assert**: we fetched a frame with a firing assert—all cycles until assertion recovery are tallied as Assert cycles; **Mispredict**: we fetched a mispredicted branch that has not yet resolved, or have encountered a BTB miss; **Miss**: FCache/ICache miss; **Stall**: some downstream execution buffer, such as the scheduling window, is full; **Wait**: turnaround cycles going from FCache to ICache fetch; **Frame**; or **ICache**.

The major impact of the micro-operation optimizer is a reduction in the number of cycles in the Frame category. The optimizer reduces the micro-operation count per frame, allowing the frame to be fetched in fewer cycles. The average net reduction in Frame cycles between the RP and RPO configurations is about 21%. This reduction also increases Frame Cache efficiency, as fewer slots are required to contain the same number of original micro-operations.

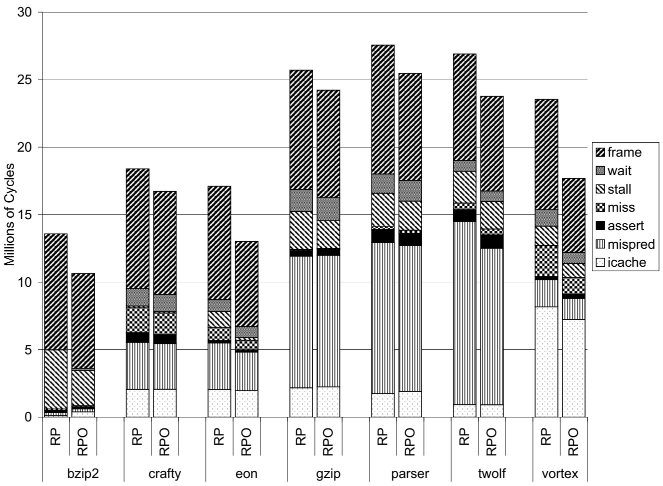

Figure 7. Per-benchmark execution cycles for the RP and RPO configurations. Each cycle is classified by the type of fetch event that occurred during that cycle.

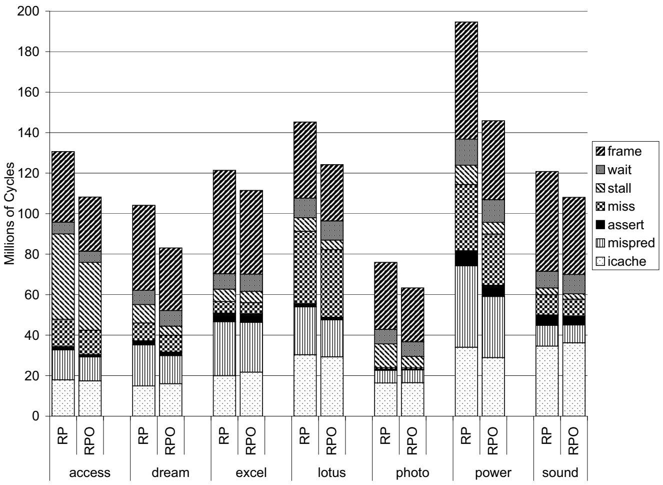

Figure 8. Per-benchmark execution cycles for the RP and RPO configurations. Each cycle is classified by the type of fetch event that occurred during that cycle.

In addition to a reduction in Frame cycles, there are other benefits to optimization. Some applications demonstrate fewer Stall cycles, such as DreamWeaver and eon. Here, the ability to represent computations more compactly results in fewer micro-operations in flight, and therefore uses fewer resources. There is also a reduction in branch resolution time, partly due to micro-operation reduction and partly due to tree height reduction. With the optimizer enabled, the branch resolution time drops by 4% and 9% on DreamWeaver and eon respectively. (keep in mind that the fetch-to-execute pipe length for branches is 15 cycles minimum). Also, it should be noted that in all benchmarks, there is a reduction in Miss cycles which accounts for an effective increase in fetch bandwidth.

Application	Micro-ops Removed	Loads Removed	Increase in IPC
bzip2	23%	30%	28%
crafty	16%	11%	10%
eon	25%	18%	31%
gzip	13%	10%	6%
parser	21%	14%	8%
twolf	14%	15%	13%
vortex	24%	34%	33%
Access	22%	20%	21%
DreamWeaver	28%	30%	26%
Excel	21%	21%	13%
LotusNotes	22%	26%	11%
PhotoShop	15%	19%	30%
PowerPoint	32%	34%	6%
SoundForge	22%	23%	6%
Average	21%	22%	17%

Table 3. The percentage of micro-operations and LOADs removed by the rePLay optimizer.

The number of cycles lost due to assertions accounts for less than 3% of execution cycles for the average benchmark, largely due to the infrequency of assertions firing or misspeculation around an unsafe store. The number of lost cycles is small despite a long assertion resolution latency. For simplicity of simulation, we only initiate recovery after *all* instructions in a frame are ready for retirement. This pessimistic model differs from branch recovery in modern processors, which is triggered when a branch executes.

6.2 Reduction in micro-operations

Table 3 shows the fraction of all dynamic micro-operations that are removed with the optimizer. On average, 21% of dynamic operations are removed. The optimizer removes 22% of all dynamic loads through the store forwarding, redundant load elimination, and dead code elimination optimizations. This result is significant both for performance and for the implications on power. As can be seen in the correlation between dynamic micro-operations removed and IPC, removing micro-operations boosts performance. Depending strongly on the construction of the optimizer, the reduction in switching activity resulting from not having to execute the removed micro-operations may also reduce power requirements.

The reduction in load micro-operations is potentially even more significant. The loads that are removed are not long-latency loads; they are likely to hit in the L1 cache because they are covered by another load or store. But those loads

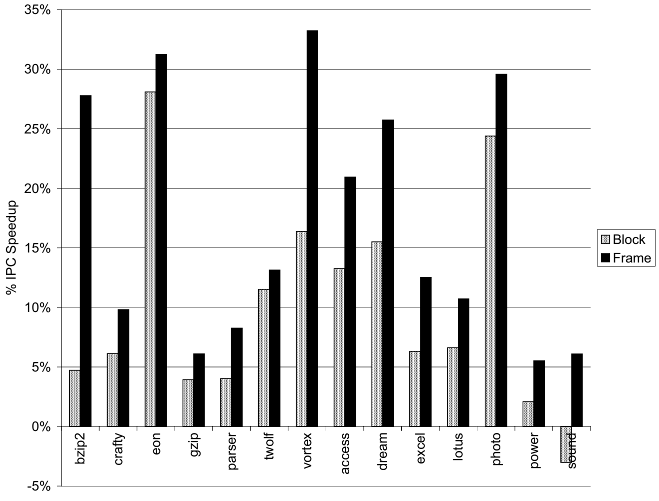

Figure 9. The percent increase in IPC when frames are optimized only within individual basic blocks versus when they are optimized as a unit.

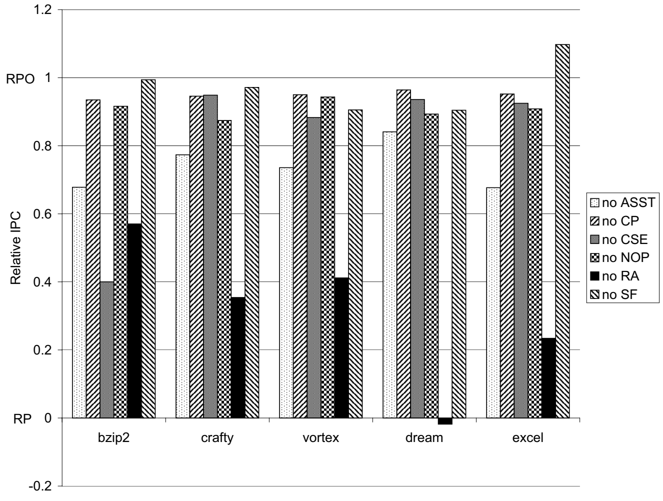

Figure 10. The performance impact of individual optimizations.

that are removed because of store forwarding are expensive because they require a store-buffer bypass. The reduction in bandwidth with fewer loads also means that the underlying pipeline can potentially be designed with fewer L1 data cache read ports or associative store buffer ports. Removing loads can have a compounded effect on power: fewer L1 data cache ports are required to sustain performance with fewer total port accesses of both the data cache and the associative store buffer.

6.3 Intra-block optimization

In order to further discern the necessary components of effective micro-operation optimization, we limited optimizations to the intra-block level on the basic blocks that constitute each frame. For example, if a frame consists of blocks A, B, and C, the optimizer optimizes A individually from B individually from C. The example in Figure 2 of Section 3 illustrates the differences with frame-level optimization.

The chart in Figure 9 contains the results of a simple experiment: measure the increase in IPC due to intra-block optimizations versus frame-level optimizations, as measured on a select group of traces from our trace set. Block-level optimizations offer some benefit, but it is not surprising to see that frame-level optimizations yield more substantial gains.The nature of the x86 ISA provides opportunities for block-level optimization even if the compiled binary is aggressively optimized by the compiler.

In the case of basic rePLay(RP), frames are immediately deposited in the frame cache upon leaving the frame constructor. If the benefit of optimizations do not outweigh the cost of delaying frames in the optimizer, then basic rePLay can outperform rePLay with optimizations. This is evident in the block-level optimization of Soundforge in Figure 9.

6.4 Impact of individual optimizations

In our final analysis, we attempt to isolate the contributions of the individual optimizations on overall performance. The optimizations individually provide little benefit, but their synergistic actions enable effective transformations.

For example, many of the optimizations leave dead code. Reassociation can change the parent of an instruction to the grandparent, potentially removing the need to execute the parent and allowing it to be removed as dead.

In this experiment, we *disabled* each optimization individually. That is, starting from a baseline of all optimizations enabled, we ran six different trials with one of the optimizations turned off. Figure 10 plots the resulting performance relative to the **RP** and **RPO** configurations. That is, 0 on the vertical scale corresponds to performance without optimizations (**RP**) and 1 on the scale corresponds to performance with all optimizations enabled (**RPO**). We disable, in sequence: (1) value ASSerTion optimization, which combines comparisons with assertions; (2) Constant Propagation; (3) Common Subexpression Elimination; (4) NOP removal, which includes removal of unconditional branches within a frame; (5) ReAssociation; and (6) Store Forwarding. As all other optimizations rely on dead code elimination, it is enabled in all runs. For clarity, we include only those applications for which optimization provides a significant performance advantage.

There is one clear trend: reassociation is a significant optimization. On DreamWeaver and Excel, IPC is nearly reduced to that of RP when reassociation is disabled. Reassociation is a gateway optimization that not only reduces tree height, but enables Common Subexpression Elimination

(CSE) to detect redundant loads and Store Forwarding (SF) to detect forwarded loads. For example, stack pointer manipulations within a frame often prevent memory optimizations from detecting equivalent addresses (during optimization two memory instructions are deemed equivalent only if their base registers are symbolically the same and their immediates and scales are literally the same). It is only after the stack manipulations have been flattened by RA that redundant and forwarded loads are detected. On the bzip2 benchmark, the effect of CSE is dominant. Here, CSE is able to detect and remove redundant loads from a critical loop within the benchmark.

Aggressive speculation during optimization can decrease potential IPC. For example, in Figure 10, Excel exhibits an increase in effective IPC when the Store Forwarding optimization is disabled. This optimization can create unsafe stores that can alias during execution. In Excel, there are many aliasing events among unsafe stores, which cause the rate of asserting frames to increase. This increase accounts for the effective IPC difference.

7 Related Work

As discussed in Section 1, this paper differs from the previous rePLay work mainly in that this work deals with the ramifications of rePLay-style optimizations for an x86 microarchitecture. We also present a detailed overview of the design of an optimization datapath. Previous rePLay work dealt with Frame Construction [13] and the application of rePLay optimizations at the Alpha ISA level.

The concept of hardware assisted dynamic optimization has evolved out of work on trace caches. The initial work on trace optimization [6, 9] dealt with simple microarchitectural optimizations (instruction fusion, instruction routing) on small trace fragments. The small trace lengths rendered compiler optimizations such as the ones considered in this paper largely ineffective for boosting performance. Subsequent work generalized the construction and optimization hardware with a separate co-processor to deal with traces [2]. The rePLay framework pushes in the direction of aggressive trace optimization by providing support for speculative optimizations on long, atomic instruction traces.

Along with rePLay, there have been other recent proposals for hardware-assisted dynamic optimization, all of which offer different perspectives on attacking similar opportunities. ROAR [12] is a run-time optimization architecture that identifies and optimizes hot regions of execution via a combined hardware and software mechanism. ROAR uses a rolling commit speculation model (rePLay uses hardware rollback) that supports interruption within an optimized region. A second approach is that of Approximate Code [17]. With this approach, executables are made up of two parts: distilled code that is optimized using assumptions about execution values, paths, etc., and the original, unmodified code. The original code executes as a separate thread to verify that the distilled code is operating correctly.

Moving upwards in abstraction layers, there have been multiple approaches to software-based dynamic optimization [1, 3, 7, 10]. For many schemes, such as Dynamo [1] and Transmeta's Code Morphing System [10], the original program runs under control of a software interpreter. The interpreter gathers information about the program's run-time behavior and builds optimized regions. When a PC is encountered for which an optimized region exists, the optimized code is directly executed.

Perhaps the most similar approach to the work described in this paper is Transmeta's Code Morphing System [10]. Transmeta's morpher is a system that transforms x86 instructions into optimized VLIW operations, similar to optimized micro-operation frames in this study. While many aspects of Transmeta's system have not been publicly disclosed, the system described in [10] is a software system that pushes the x86 architectural boundary to the basic block level, allowing intra-block optimizations. This approach thus has higher overhead and performs less aggressive optimization than is possible with rePLay, as illustrated by the example in Section 3.

8 Conclusion

In this paper we demonstrate that complex instruction set architectures contain inefficiencies that do not exist in simpler ISAs. Using the rePLay Framework as the architectural substrate, we were able to execute a series of optimizations that exploit those opportunities that a compiler cannot take advantage of due to ISA barriers. Simulation results reveal that these optimizations drastically reduce both the total number of micro-operations and the number of load micro-operations across a set of representative benchmarks from both SPECint 2000 and desktop applications. These reductions contribute to a boost in performance on a rePLay-enabled superscalar processor.

9 Acknowledgments

We thank the other members of the Advanced Computing Systems group, as well as Stephan Jourdan, Ben Sander, and Jared Stark for providing feedback during various stages of this work. This material is based upon work supported by the National Science Foundation under Grant Nos. 0092740 and 9984492 and the C2S2 Marco Center, with very gracious support from AMD, Intel, and Sun.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[2] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[3] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[4] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[5] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[6] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.

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

[8] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming*, 26(4):449–478, Aug. 1998.

[9] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[10] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corporation, Jan. 2000.

[11] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, June 1995.

[12] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[13] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, 2000.

[14] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, 1988.

[15] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 143–147, 1994.

[16] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, 1997.

[17] C. Zilles and G. Sohi. Master/slave speculative parallelization with distilled programs. In *Wisconsin Madison Technical Report TR-1438*, 2002.