Using Dynamic Binary Translation to Fuse Dependent Instructions

Shiliang Hu
Dept. of Computer Sciences
University of Wisconsin - Madison
shiliang@cs.wisc.edu

James E. Smith
Dept. of Electrical & Computer Engineering
University of Wisconsin - Madison
jes@ece.wisc.edu

Abstract

Instruction scheduling hardware can be simplified and easily pipelined if pairs of dependent instructions are fused so they share a single instruction scheduling slot. We study an implementation of the x86 ISA that dynamically translates x86 code to an underlying ISA that supports instruction fusing. A microarchitecture that is co-designed with the fused instruction set completes the implementation.

In this paper, we focus on the dynamic binary translator for such a co-designed x86 virtual machine. The dynamic binary translator first cracks x86 instructions belonging to hot superblocks into RISC-style micro-operations, and then uses heuristics to fuse together pairs of dependent micro-operations. Experimental results with SPEC2000 integer benchmarks demonstrate that: (1) the fused ISA with dynamic binary translation reduces the number of scheduling decisions by about 30% versus a conventional implementation that uses hardware cracking into RISC micro-operations; (2) an instruction scheduling slot needs only hold two source register fields even though it may hold two instructions; (3) translations generated in the proposed ISA consume about 30% less storage than a corresponding fixed-length RISC-style ISA.

1. Introduction

Two basic techniques for improving processor performance are increasing instruction level parallelism (ILP) and increasing pipeline depth (thereby achieving a higher clock speed). Each of these is a challenge on its own, and the two techniques are often at odds with each other. For example, attempting to extract higher ILP tends to increase the size of the instruction scheduling window, which makes a deeper pipeline more difficult. In addition, the practical problems of implementing a CISC instruction set like the Intel x86 (aka IA-32) introduces additional performance challenges. For example, splitting or "cracking" an x86 instruction into a number of RISC-like micro-operations tends to increase the number of operations that must be individually issued in order to execute the original program.

We are studying a co-designed virtual machine implementation[17][18][19] of the x86 instruction set that uses an underlying implementation instruction set which permits combining or "fusing" pairs of dependent instructions. These fused instructions share a scheduling window slot and are scheduled by hardware as a unit, i.e. as if they were a single instruction. However, the instructions actually begin execution in successive cycles, as in a conventional superscalar processor. Because of the dependency, a fused pair of instructions takes at least two clock cycles to execute. Furthermore, after instruction fusing is done, many of the remaining non-fused instructions also take two or more cycles to execute. Under these circumstances, i.e. two (or more) cycles of execution latency, the hardware scheduling logic can be pipelined both simply and with relatively little performance loss. The wakeup and select functions can be performed in two separate clock cycles without relying on speculation. Thus, the use of a fused instruction set *1) allows ILP to be increased without adding scheduling window slots and 2) enables simple pipelined scheduling logic that is conducive to deeper pipelining.*

1.1. Illustrative Example

A snippet of x86 code taken from benchmark 176.gcc is shown at the left of Figure 1. This code sequence contains 15 x86 instructions. A straightforward cracking will translate the code sequence into 21 RISC style micro-ops. However, if pairs of dependent x86 RISC-ops are combined into fused instruction pairs (shown on the right side of the figure), then the same sequence consumes only 13 scheduling window slots in the dynamic processor core.

	X86 instructions	Fused ISA	Execution Latency
1	mov ebx,ds:[esi + 1c]	LD Rebx, [Resi + 1c]	3
2	test ebx, ebx	TEST Rebx, Rebx :: Jz 126	2
3	jz 08115bf2		
4		LD Rtmp, [Rebx + 02]	3
5	cmp ds:[ebx + 02], 0d	CMP Rtmp, 0d :: Jz 2f	2
6	jnz 08115ae1		
7	mov ebx,ds:[ebx + 08]	LD Rebx, [Rebx + 08]	3
8	test ebx, ebx	TEST Rebx, Rebx :: Jnz e4	2
9	jnz 08115bcc		
10	jmp 08115bf2	(direct jmp removed)	
11	add esp, 0c	ADD.cc Resp, 0c :: LD Rebx,[Resp]	4
12	pop ebx	ADD Resp, 4 :: LD Resi,[Resp]	4
13	pop esi	ADD Resp, 4 :: LD Redi,[Resp]	4
14	pop edi	ADD Resp, 4 :: LD Rebp,[Resp]	4
15	pop ebp	ADD Resp, 4 :: LD Rtmp,[Resp]	4
16	ret_near	ADD Resp, 4	1
17		BR.ret Rtmp	1
	37 Bytes 15 x86 instructions	50 Bytes, 21 RISC-like instructions. Consume 13 scheduling window slots	

Figure 1. Example from SPEC2000-INT 176.gcc

Instruction formats in the proposed implementation ISA may be either 16 or 32 bits long. The instruction in the first line of the example in Figure 1 uses the long, 32-bit format. Lines 11 through 15 contain two fused 16-bit instructions. The two fused instructions are shown on the same line, separated by double colons "**::**". The first instruction in the fused pair is defined to be the *head* and the second is the *tail*.

Finally, the number at the end of each line in the figure is the minimum number of cycles that each corresponding instruction (or fused pair) will take for execution. Virtually all of them are two cycles or more. The only exception is the last two lines, which have two independent instructions that execute in one cycle each.

1.2. Dynamic Binary Translation for x86

In the co-designed VM implementation we are studying, dynamic binary translation maps x86 instructions into the fused instruction set. We note that the native x86 instruction set already contains what are essentially fused operations. For example, the x86 instruction `add eax,[ebp+4]` performs a load and an arithmetic operation. However, the fused instruction set we propose allows some forms of operation pairing that the x86 does not allow, for example the proposed instruction set can fuse a condition test operation followed by a conditional branch, or it can fuse two simple ALU operations. Furthermore, the optimization heuristics we use often fuse operations in different combinations than in the original x86 code.

Dynamic binary translation/optimization using a code cache held in main memory has a number of advantages. For example, superblock [23] formation, as is typically done, leads to improved spatial locality in the instruction stream. Because software is being used for optimization, relatively sophisticated algorithms can be used; this is the case in the DAISY [18] and Transmeta [17] VLIW machines. Finally, because optimization is being done dynamically, profile-directed optimizations can be done transparently, using profile data from the program being optimized.

On the other hand, if one uses software dynamic binary translation from a dense instruction set such as the x86 to conventional RISC-style instructions, there are significant performance disadvantages due to the resulting code expansion. That is, the RISC-style instructions will consume more memory space, causing reduced instruction cache performance. In addition, the instruction fetch bandwidth would be used much less efficiently than with the original x86 code. Consequently, such an approach is at a considerable disadvantage when compared with conventional hardware cracking where x86 instructions are fetched from main memory and RISC-ops are formed in the instruction pipeline [11][12][13][14].

The proposed fused ISA at least partially overcomes the code density disadvantage because it permits a denser encoding than a conventional RISC ISA. Frequently, two short instructions are combined into a single 32-bit word. For example, in Figure 1, the original x86 code consumes 37 bytes of storage. If this code were translated into RISC instructions of 32-bits each, then a translated RISC version consumes 84 bytes. In contrast, with our 16/32-bit variable length instructions, the program consumes 50 bytes.

1.3. Related Work

The motivation for the proposed co-designed VM implementation springs from earlier work on dependent instruction strands [4] and on a recent microarchitecture proposed by I. Kim and M. Lipasti [3]. The work on instruction strands is targeted at a microarchitecture that exploits the natural dependences in a program by issuing dependent sequences (strands) from simplified FIFO issue queues. However, most sequences of dependent instructions tend to be rather short, often only two or three instructions. Consequently, Kim and Lipasti proposed *hardware* fusing of RISC instructions (i.e. Alpha) to consume single scheduling window slots. They made the key observation that because dependent strands are short, then building a microarchitecture that exploits only dependent *pairs* can yield most of the performance advantages.

An important aspect of the approach presented here, which sets it apart from the Kim and Lipasti approach, is the use of the co-designed paradigm with software generation of fused instruction pairs. This removes considerable complexity from the hardware and enables more sophisticated fusing heuristics. Reducing hardware complexity is especially important when starting with a complex instruction set (a second important difference from the method described in [3], which uses a RISC ISA).

Some recent x86 implementations have gone in the direction of more complex internal operations, at least for certain stages of the pipeline. The AMD Opteron [14] uses a Macro-Operation designed to reduce the dynamic operation count. In most common cases, one x86 instruction is mapped to one internal Macro-Operation. The Intel Pentium M microarchitecture [12] fuses memory store and load operations with the address arithmetic operation from the same original x86 instruction. This is intended to reduce instruction traffic through the pipeline for performance and energy efficiency. The operations in each pair are still scheduled separately. Our proposed method distinguishes itself from these designs by using software fusing of suitable operations that may cross original x86 instruction boundaries. In fact, as will be shown, more than half of the fused instructions contain operations from different x86 instructions.

The IBM POWER4 micro-architecture [15] forms 5-slot groups of instructions to reduce instruction-tracking overhead. The slots in each group have fixed assignments. This method does not reduce the total number of slots in the instruction window or the number of instructions that are issued.

The Transmeta Crusoe processor [17] and the IBM DAISY [18] are co-designed VMs that have an internal VLIW-style instruction set, composed of RISC-like operations. With the VLIW approach, considerable software optimization is required for reordering instructions, especially if speculation is embedded in the instruction set. In contrast, the underlying hardware we envision is fully capable of dynamic instruction scheduling; the primary software function is superblock formation and instruction pairing. Hence, we anticipate significantly simpler translation software than in the VLIW implementations.

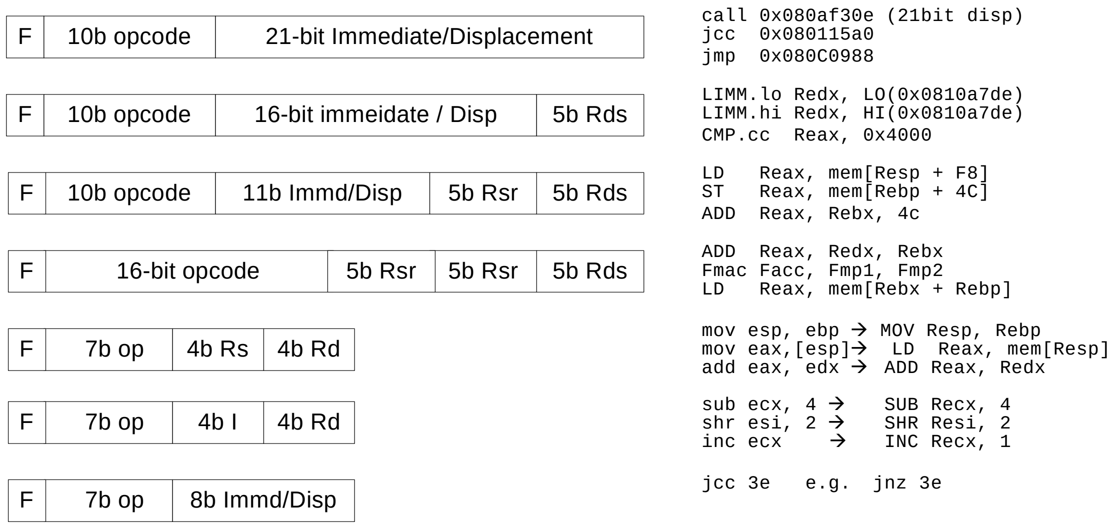
Methods for implementing fast scheduling logic have been published recently. A pipelined solution using speculation was proposed in [1]. This method is based on dynamic detection and tracking of dependence chains, and, as with most speculative schemes, it needs recovery mechanisms which tend to complicate the scheduling logic overall. Other papers have proposed ways of reducing the scheduling logic latency [6][7] or pre-scheduling to enable an effectively larger issue window with a small physical issue window [8][9][10].

With regard to dependence-based instruction set design, an unimplemented design for a Cray Research processor [16] packed multiple dependent operations together with an implicit accumulator to chain them together. The research cited above by H.-S. Kim and Smith [4] proposed an ISA that chains dependent instructions together with an explicit accumulator specifier.

1.4. Paper Overview

There are two major components to a co-designed VM implementation – the dynamic binary translator and the supporting microarchitecture. Many of the significant microarchitecture issues have been explored by Kim and Lipasti [3]. It is our objective in this paper to explore the feasibility of (1) using dynamic software translation to form fused instruction pairs and (2) applying the method to a complex instruction set, the x86. Hence, we focus on the dynamic binary translation aspect with the goal of maximizing the number of fused instructions presented to the hardware. We also study properties of the translated x86 code to serve as a guide for possible fused instruction set revisions and to guide microarchitecture design.

Section 2 describes an instruction set supporting fused operations that is designed to implement the x86 ISA. Section 3 describes aspects of the microarchitecture that are necessary for understanding instruction set features and translation optimization heuristics. Section 4 presents a dynamic binary translator that cracks x86 instructions into micro-operations, and then fuses and optimizes them for the envisioned co-designed processor. Preliminary dynamic binary translator design evaluation and program characterization results are presented in section 5. Section 6 concludes the paper.


F | 10b opcode | 21-bit Immediate/Displacement
F | 10b opcode | 16-bit immeidate / Disp | 5b Rds
F | 10b opcode | 11b Immd/Disp | 5b Rsr | 5b Rds
F | 16-bit opcode | 5b Rsr | 5b Rsr | 5b Rds
F | 7b op | 4b Rs | 4b Rd
F | 7b op | 4b I | 4b Rd
F | 7b op | 8b Immd/Disp

```
call 0x080af30e (21bit disp)
jcc  0x080115a0
jmp  0x080C0988

LIMM.lo Redx, LO(0x0810a7de)
LIMM.hi Redx, HI(0x0810a7de)
CMP.cc  Reax, 0x4000

LD   Reax, mem[Resp + F8]
ST   Reax, mem[Rebp + 4C]
ADD  Reax, Rebx, 4c

ADD  Reax, Redx, Rebx
Fmac Facc, Fmp1, Fmp2
LD   Reax, mem[Rebx + Rebp]

mov esp, ebp → MOV Resp, Rebp
mov eax,[esp]→  LD  Reax, mem[Resp]
add eax, edx → ADD Reax, Redx

sub ecx, 4 →    SUB Recx, 4
shr esi, 2 →    SHR Resi, 2
inc ecx    →    INC Recx, 1

jcc 3e   e.g.  jnz 3e
```

Figure 2 Proposed Instruction Set

2. A Fused Instruction Set for x86

We first describe the architected register state of the proposed fused instruction set, and then describe the instruction formats with example instructions.

2.1. The Architected State

The architected register file has the following state:

- 32 general-purpose registers, R0 through R31, each 32-bit wide. Reads to R15 always return a zero value and writes to R15 have no effect on the architected state.
- 32 floating-point, MMX/SSE SIMD extension registers, F0 through F31, each 128 bits wide.
- The program counter and condition code (x86 EFLAGS) registers.
- System-level or special registers.

2.2. Instructions

The proposed instruction set (Figure 2) contains instructions that are in either a long 32-bit format or a short 16-bit format. In this study, we explore both the case where 32-bit instructions can span two words and the case where 32-bit instructions are forced to be aligned on 32-bit boundaries with no-op padding when necessary. The latter approach may simplify the hardware implementation.

The 32-bit formats can encode general 3-operand instructions or instructions that need long immediate values. The 16-bit formats are intended to increase code density and use an x86-like 2-operand format in which one of operands is both a source and a destination. All short instructions have a corresponding long format. Note that short format operations can only access the lower 16 registers. However, this does not pose any significant problems for our co-designed VM scheme. Shown next to each format in Figure 2 are example instructions that can be encoded in the format.

The first bit of each instruction indicates whether the instruction should be fused with the immediately following instruction. That is, whether the instruction dispatch hardware should assign both instructions to the same scheduling window slot.

There are three addressing modes in the co-designed ISA; the formats are chosen to match the important x86 addressing modes.

- Register indirect addressing: ***mem[register];***
- Register displacement addressing: ***mem[register + 11b_displacement]***, and
- Register indexing addressing: ***mem[Ra+(Rb<<shmt)]***. This mode takes a 3-register operand format and a shift amount, from 0 to 3 as used in the x86.

In the instruction formats shown in Figure 2, opcode and immediate fields are adjacent to each other to highlight a potential trade-off of the field lengths; i.e. the opcode space can be increased at the expense of the immediate field and vice versa.

Finally, x86 exceptions and interrupts are mapped directly into the co-designed implementation ISA.

3. Microarchitecture Overview

As noted above, the focus of the study presented here is not on the microarchitecture of the proposed co-designed VM; the goal is to explore the issues that affect the instruction set and dynamic binary translation. In this section we briefly overview only those microarchitecture features that are important for understanding the software translation methods and instruction set features.

Overall, the microarchitecture we envision is very similar to that used in a conventional out-of-order superscalar processor. The main difference is in the way the instruction scheduler operates.

3.1. Scheduling Window Management

There are two relevant aspects to the scheduling window implementation. The first is the window slot contents, and the second is the pipelining of scheduling logic.

Because two fused instructions are dispatched to the same scheduling slot, the slot ostensibly needs more register specifiers than in a conventional single instruction slot. That is, to include information for two instructions, the slot could contain up to four source registers and two destination registers. Because renaming is used, the second destination register probably does not cause significant delays in the instruction scheduling logic beyond those for a single instruction, but any additional source register(s) may. Data to be shown later, however, indicates that fused instruction pairs with four source registers are very rare, and for the significant majority of cases, an implementation with only two source registers per scheduling slot will suffice.

Turning to the topic of scheduler pipelining, in order to schedule (and issue) a pair of fused instructions, all source operands for the pair must be available. Then, the pair of instructions takes at least two cycles to execute. Because the total execution latency is at least two cycles, scheduling logic wakeup and select functions can be done in separate pipeline stages without significant performance loss. That is, the cycle after a two-cycle instruction is scheduled to start, its completion can be anticipated and dependent instructions can be awakened. Then, these newly awakened instructions have a full second cycle in which they can be selected before their input data is known to be available.

Note that the microarchitecture does not necessarily require modifications to the register file or functional units. That is, even though a fused pair is *scheduled* as a unit, the two halves of the pair can actually *begin execution* in two different (consecutive) clock cycles. We do not assume fused functional units [24], and one scheduling decision kicks off two consecutive, serialized issues from the same slot.

3.2. Performance Implications

Clearly, using scheduling window slots more efficiently and pipelining the scheduling logic have performance advantages, primarily in allowing a faster clock cycle for a given level of ILP. The performance disadvantages will take the form of an increased number of clock cycles for the following reasons.

First, the result of the head instruction of a pair is made immediately available to the tail instruction, but all other instructions using the value produced by the head must effectively wait an extra cycle. Because most data values are consumed by only one instruction [4], this will be a relatively uncommon occurrence.

Second, source operands for both instructions of a fused pair must be available before the head instruction can begin execution. Data given in [3] by I. Kim and Lipasti indicate that this is also fairly uncommon.

Third, any instructions data dependent on a *non-fused* single-cycle instruction may suffer a penalty due to the two-cycle, pipelined scheduling logic. Here, we use heuristics that attempt to maximize the number of single-cycle instructions that are fused. Consequently most of the leftover non-fused instructions either do not produce register values or tend to take multiple execution cycles. Data to be given later show that there are few single-cycle, non-fused instructions that produce register values.

4. Dynamic Binary Translation

The task of our dynamic binary translator is to crack x86 instructions from a hot superblock into a RISC-like intermediate form and then perform instruction fusing and other optimizations directed at the co-designed processor. We first discuss how native registers are allocated and then describe the binary translation steps. Next, we describe instruction fusing algorithms for the envisioned micro-architecture. We note that although the translator re-orders instructions, the re-ordering is done only for the purpose of fusing dependent instructions. The ordering between load and store operations is strictly maintained as in the original x86 instruction sequence.

4.1.State Mapping and Long Immediate Values

To emulate the x86 ISA efficiently, a permanent register state mapping is used. The eight x86 general-purpose registers are mapped to the first eight of the 32 general purpose integer registers. This mapping is maintained at superblock boundaries. For the first eight registers, we use an x86-like notation for readability (e.g. Reax corresponds to x86 eax; this was done in Fig. 1). Registers R8 to R14 are temporary registers and are mostly used for providing local communication between two operations cracked from the same x86 instruction. R15 is the zero register; because x86 binaries have a large number of zero immediate values this zero register can reduce dynamic instruction count considerably.

Registers R16 to R31 are used mainly by virtual machine software for x86 interpretation, binary translation, code cache management, precise state recovery, etc. Using a separate set of registers for VM software can avoid the overhead of "context switches" between the VM code and the translated native code. As a consequence, the VM code often uses long format instructions rather than the short equivalents.

Due to the small number of general-purpose registers in the x86 ISA, x86 binaries tend to have more immediate values than a typical RISC binary. For example, memory addressing via absolute addresses embedded in x86 instructions occurs in about 10% of all x86 memory access instructions [5]. These 32-bit long immediate values are problematic when translating to an instruction set with maximum-length 32-bit instructions. A naïve translation would use extra instructions to build up every long immediate value. To reduce the code expansion that would result, the proposed binary translator collects these long immediate values, analyzes the deltas among values to find values that "cluster" around a central value, and then converts a long immediate operand to: (1) a register operand, if the value has already been loaded or (2) a register operand plus or minus a short immediate operand, if the immediate value falls within a certain range of an already registered immediate value. In this manner, an absolute addressing instruction can often be converted to a register indirect or register displacement addressing instruction.

Experimental results will show that four or five registers are usually sufficient for holding immediate values used by a superblock, and some or all of these can be allocated to registers R11 through R14 to facilitate register indirect addressing in the short format. Currently, this transformation is limited to within single superblocks; in future research we will study more global methods for converting long immediates to register values.

4.2. Translation Method

We use a hot superblock detection and formation algorithm that is a slightly modified version of Dynamo's Most Recently Executed Tail (MRET) heuristic [20]. Unlike Dynamo, our translator stops constructing a superblock when an indirect jump is encountered. We use a maximum superblock size of 256 RISC-like operations cracked from x86 instructions and a usage counter threshold of 32. The algorithm treats an x86 string instruction with a repetition prefix as a separate superblock that forms a tight loop. After a hot superblock has been formed, the dynamic binary translator performs the following steps.

1) The x86 instructions are cracked into abstract RISC-like micro-operations. Abstract register-to-register and register-short immediate micro-operations are essentially the same as the final generated instructions. Memory access instructions and instructions with embedded long immediate values are transformed into abstract micro-operations that preserve the logical semantics of the original x86 instructions.

2) The superblocks are scanned for long immediate values, value clustering analysis is performed, and immediate values are allocated to registers as described in the previous subsection.

3) The abstract micro-operations are transformed into instructions belonging to the fused instruction set described in Section 2.

4) Instruction fusing is performed. After a conventional dependence analysis and setup of the dependence chains, dependent instructions are paired together to form fused instructions (more detail on the fusing algorithms is given below). Dependent pairs are not fused across conditional branches (and indirect jumps, implied by superblock formation). However, dependent instructions across direct jumps or calls can be fused. Condition codes in the x86 ISA are handled as normal data dependences and many fused pairs are in fact formed around condition codes.

5) Register allocation is performed. Note that before this step all register numbers are pseudo register numbers. The issue here is that in order to allow precise state recovery as described in Le [22], physical register allocation has to be done at this point. As instructions are reordered, register live ranges are extended to allow precise state recovery. Permanent register state mapping is maintained at all superblock boundaries. As we will see in the evaluation section, most fused instructions are consecutive or are at least very close to their

locations in the original sequence cracked from x86 instructions. Consequently, occasional re-ordering of translated code does not lead to excessive extra register copies for restoring the state mapping at the end of a superblock.

6) Code is generated: The fused ISA instructions with fusing information are generated and the translated code is added to the code cache for native execution.

4.3. Fusing Algorithms

The objectives of the dynamic instruction fusing algorithms are 1) to maximize the number of fused dependent instruction pairs (to optimize usage of scheduling window slots) and 2) to minimize the number of single cycle instructions that are not paired (to reduce performance losses from pipelined scheduling logic). A number of heuristics are used. One heuristic is to always use a single-cycle instruction as the head instruction of a pair. A multi-cycle instruction will not see performance losses from pipelined scheduling logic, so there is relatively little value in using it as a head instruction. A second heuristic is to first try to pair instructions that are close together in the original x86 code sequence. The rationale here is that these pairs are more likely to be dependent instructions that need to be scheduled for back-to-back execution in order to reduce the program's critical path. Consecutive or close pairs also tend to be less problematic with regard to other issues, e.g. register live ranges need to be extended less in order to provide precise state recovery.

We concentrate on fusing algorithms that can be performed as a single-pass scan in order to enable fast dynamic translation. We consider two possibilities, one that does a forward scan and one that does a backward scan. After the construction of the data dependence graph, a *forward scan* algorithm considers instructions one-by-one as candidate *tail* instructions. That is, for each potential tail, it looks backwards in the instruction stream for a head. It does this by scanning from the second instruction to the last instruction in the superblock attempting to fuse each not-yet-fused instruction with the nearest preceding, not-yet-fused single-cycle instruction that produces one of its input operands. A backward scan algorithm traverses from the second last instruction to the first instruction, and considers each instruction as a potential *head* of a pair. Each not-yet fused single-cycle instruction is fused with the nearest not-yet-fused consumer of its generated value. Note that the direction of searching for a fusing candidate in these algorithms is always opposite to the scan direction, and we call this an *anti-scan (direction) fusing heuristic*. The rationale for this will be seen in subsection 4.4.

Neither the forward nor the backward scan algorithm in the dynamic binary translator is necessarily optimal. However, we believe they are near-optimal, and in cases we have manually inspected, they capture well over 90% of those possible pairs.

To explore instruction-fusing algorithms further, we also studied a more complex, iterative algorithm that scans the superblock multiple times, beginning by looking for pairs with fusing distance one, i.e. consecutive pairs. It then looks for distance two pairs, distance three pairs, etc. until all fusible instructions have been fused or until the full superblock size has been reached. Note that distances are measured with respect to locations in the original micro-operations cracked from the x86 binary. Compared to the single-pass scan algorithms, the iterative algorithm performs slightly better but it has much more runtime overhead due to its multiple scan passes.

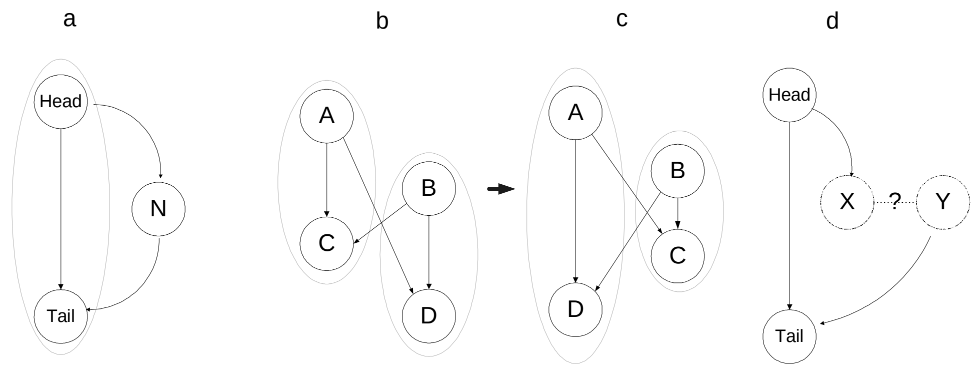


Figure 3: Dependence Cycle Detection

4.4. Dependence Cycle Detection

Although it was not mentioned above, certain data dependence patterns inhibit code re-ordering during fusion. For example, consider the case where a head candidate for a given tail produces a value for both its tail instruction and another instruction that separates them in the original cracked code sequence. If the instruction in the middle also produces an operand for the tail (Figure 3a), then making the tail and head consecutive instructions (as is done with fusing) must break one of the dependences between the candidate pair and the "middle" instruction. Note that in Figure 3, the vertical position of each node shows its order in the original sequence cracked from the x86 code. For example, A precedes B; C precedes D and so on. Other situations where data dependences can prevent fusing involve cross dependence of two candidate pairs (Figures 3b and 3c). However, analysis of these cases is not as straightforward as in figure 3a. Thus, we need an algorithm to avoid breaking data dependences in the data dependence graph.

A property of the anti-scan fusing heuristic is that it assures that the pairing shown in Figure 3b does not occur. In the case shown in Figures 3b and 3c, the algorithm will first consider pairing C with B rather than D with B, because B is the nearest operand producer for C. Consequently, the only pairing considered is the one shown in Fig 3c.

Now, there is an advantage to reducing cross dependences to the case in Figure 3c versus the case in Figure 3b. That is, while considering candidate instructions for pairing, we only need to consider instructions between the candidate head and tail for potential dependence cycles. Note that according to the anti-scan fusing heuristic, B and C are paired first for either the forward or backward scan method before A and D are considered. In contrast, one has to analyze dependent instructions either before the head or after the tail if the case shown in Figure 3b can occur.

The general case for detecting dependence cycles, of which Figure 3a and 3c are special cases, is modeled in Figure 3d. Under the anti-scan fusing heuristic, the data dependence cycle detection algorithm only needs to consider nodes between the candidate head and tail when looking for potential cycles (which inhibit fusing).

5. Evaluation

The experimental infrastructure is a full system x86 virtual machine, *x86vm*, we are currently developing. It consists of three major components: (1) An x86 interpreter/functional simulator extracted from open source BOCHS 2.0.2 [21]; (2) A dynamic binary translator that cracks x86 instructions into RISC-like micro-operations and optimizes them as described in the previous section; (3) A microarchitecture simulator. We present program characterization and dynamic binary translation results with SPEC2000 integer benchmarks. Benchmark binaries are generated by Intel C/C++ v7.1 compiler with –O3 SPEC2000 base optimization options (Binaries generated by GCC show similar results). Note that due to the lower precision of floating-point instructions (64-bits instead of 80-bits as in IA-32), simulation of some integer benchmarks (175.vpr, 252.eon, 300.twolf) does not generate exactly the same results as Intel processors.

Preliminary profiling data indicate that overhead for our dynamic binary translator is about one to two microseconds per x86 instruction translated on a 1.80GHz Pentium 4 desktop. However, this version of the dynamic binary translator is written in C++ for readability and flexibility rather than for performance. Substantial performance improvement is anticipated for a product dynamic binary translator, for example, by merging some passes described in subsection 4.2 and by coding it in highly optimized native assembly.

5.1. Superblock Size

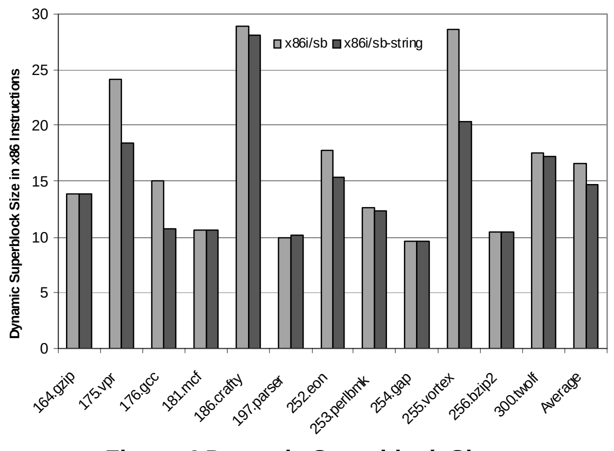


Figure 4 Dynamic Superblock Size

First, we give some basic results regarding superblock size. Although not strictly related to the main topic of the paper, these are important for assuring good performance in a code cache implementation. In Figure 4, the left bars show the average number of x86 instructions per dynamic superblock if x86 repetitive string instructions are not treated as separate superblocks. The right bars show superblock size if they are treated as separate superblocks. On average, about 15 x86-instructionequivalent operations are executed per superblock. The separate superblocks for string instructions bring down the average number, but they do not break the requirement that the first instruction in the superblock is the only entry point. A better solution might embed the string loops inside an outer superblock and perform no dependence analysis across string loops.

5.2. Immediate Conversion

As discussed in Section 4, we exploit long immediate value conversion to reduce instruction count. There are two types of immediate values in x86 instructions, immediate operands and displacements for address calculation. Because program code and data sections are usually clustered together, displacement values are more amenable to our long immediate conversion algorithm. As shown in Figure 5**.** More than half of the long displacements are successfully converted. As mentioned before, about 10% of x86 addresses are generated from absolute addresses embedded in x86 instructions, so this conversion notably reduces the code expansion associated with cracking x86 instructions into our fused instruction set. On the other hand, long immediate operands are in general much harder to convert. Fewer than 10% can be converted, which reduces the aggregate conversion rate for all long immediate values to less than 40%.

The number of temporary registers needed for holding long immediate values is moderate. In most benchmark programs, 4 or 5 registers will be sufficient to handle more than 99% (*y-axis* in Figure 6) of the dynamic superblocks. Only in some rare cases, e.g. 186.crafty, 8 or more registers are actually needed as shown in Figure 6.

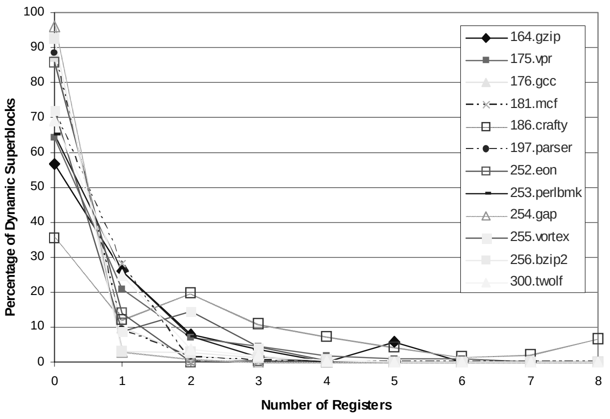


Figure 6 Number of Registers Required for Holding Long Immediate Values

5.3. Instruction Density

An ISA with good coding density can reduce instruction fetch bandwidth and improve I-cache performance. In this experiment, we assume an unbounded code cache to exclude replacement, and then compare the total translation size. Results are shown in Figure 7. The first bar for each benchmark is the normalized code size for original x86 code organized into the same superblocks as used during translation. All the bars of each set are normalized with respect to the first one. The second bar shows the total code size for the proposed 16/32-bit fused ISA. On average, the translated benchmarks consume about 33% more space than the x86 code. On the other hand, if a fixed-length (32-bits) conventional RISC-style ISA is used (the fourth bar), the expansion is from 60% to over 100%. If the 16/32-bit instruction set is used, but with alignment of 32-bit instructions to 32-bit instruction word boundaries (with no-op padding when needed), about 9% additional storage is required as shown by the third set of bars in Figure 7.

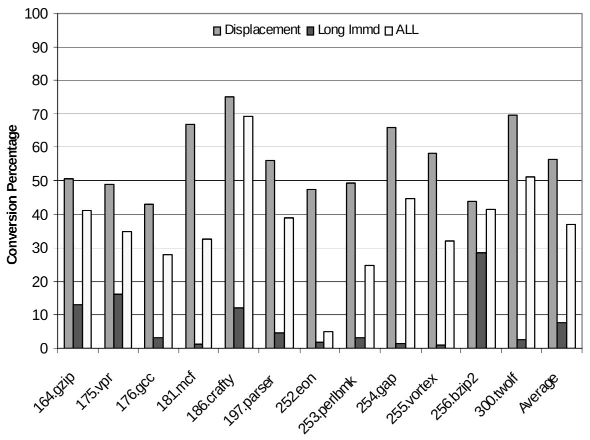


Figure 5 Percent Long Immediate Values Converted

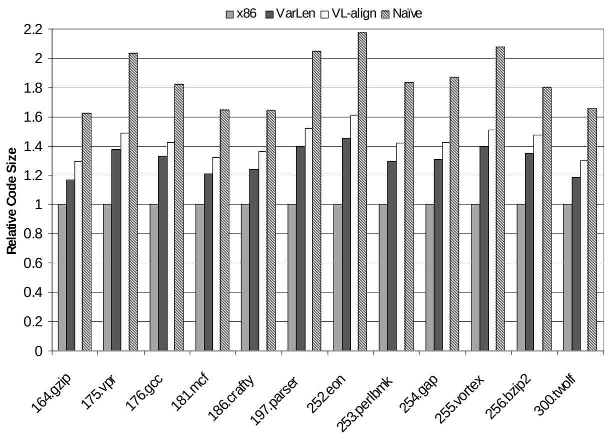


Figure 7 Static Translation Size Comparison

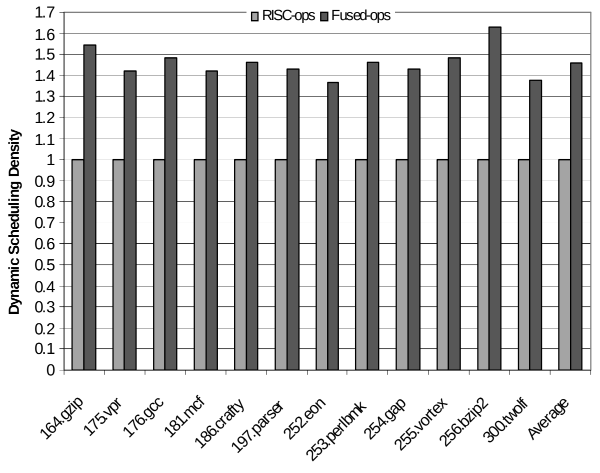


Figure 8 Dynamic Scheduling Density

We define the *dynamic scheduling density* of an ISA as the average number of RISC-like operations handled by each scheduling window slot. We compare dynamic scheduling density between x86 code cracked into RISC micro-operations and code using the fused ISA (which essentially fuses the same RISC micro-ops). Because most x86 superscalar processors allocate one scheduling window slot for each RISC operation, the x86/RISC model has a dynamic scheduling density of 1.0 by definition. With the fused ISA, each scheduling slot handles on average 1.4 ~ 1.5 RISC operations (Figure 8). Hence, one could potentially reduce the number of scheduling slots by about 30% and achieve the same instruction level parallelism as with a conventional x86 implementation (verification of this awaits future micro architecture simulation).

5.4. Fusing Heuristics

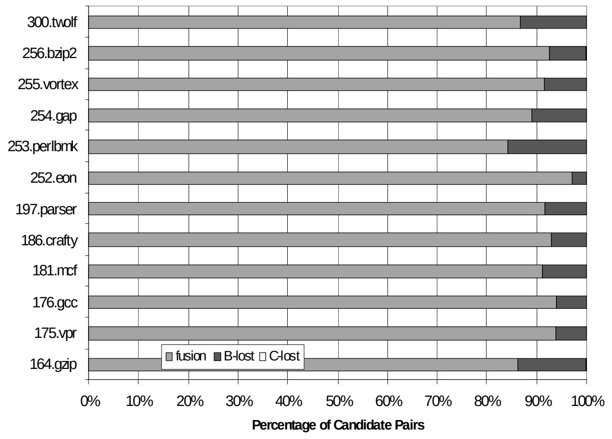


Figure 9 Instruction Fusing Profile

We now characterize x86 binary fusing properties and evaluate the effectiveness of the fusing heuristics. In this experiment, we name two instructions as a *candidate pair*, if: (1) neither of them has been fused with other instructions; (2) there is dependence between them; and (3) the head is a single-cycle instruction. After a candidate pair is found, it must pass two main fusing criteria before it is actually fused. The fusing criteria are 1) whether there is a conditional branch between the head and the tail, 2) whether there is a dependence cycle in the data dependence graph.

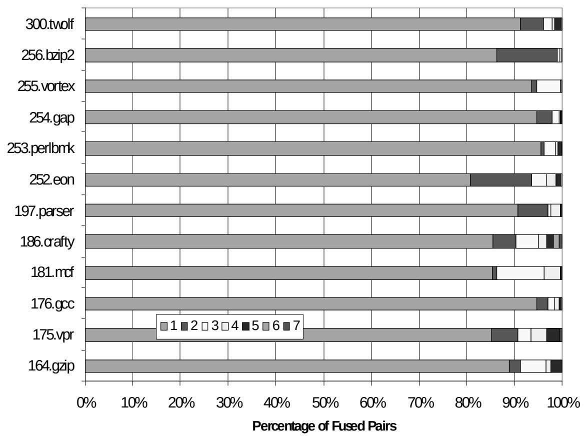


Figure 10 Head/Tail distance distribution

As shown in Figure 9, over 90% of the candidate pairs found in the data dependence graph are successfully fused. The major reason for pairs failing to fuse is that they are separated by conditional branches. These are the B-lost segments in Figure 9, and they amount to less than 10% of the candidate pairs. The pairs failing to fuse due to cycles in data dependence graph (C-lost in Figure 9) are negligible. Figure 10 shows the distance distribution of the head and tail in terms of the original micro-operation sequence cracked from the x86 binary. Here the single-pass forward scan algorithm is used. More than 80% of the fused pairs are consecutive micro-operations in the original sequence. Slightly more pairs are consecutive if the iterative pairing algorithm is used. Although it is not shown, the forward scan algorithm performs slightly better (~1% in terms of fused pairs) than the backward scan.

5.5. Code Reorganization

More than half of the fused instructions are composed of two micro-operations from different original x86 instructions (X-fuse bars in Figure 11). This implies that instruction fusing reorganizes the executable binary for the underlying co-designed hardware, rather than performing a simple reverse of the initial x86 cracking. Figure 11 also shows that, on average, more than 60% of paired instructions are single-single cycle pairs (S-fuse bars). As one may infer, our data shows that for all of the instructions translated and placed into the code cache, about 65% of them are paired together for allocation as a single entry for the scheduling logic. Among those non-fused instructions, memory access instructions and branches take nearly 80% as shown in figure 12. ALU operations are 23%. As there are about 35% non-fused instructions in the code cache, this implies that single-cycle non-fused ALU instructions make up about 8% of instructions. This tends to back up our earlier assertion that single cycle non-fused ALU instructions will cause relatively little IPC loss when scheduling logic is pipelined.

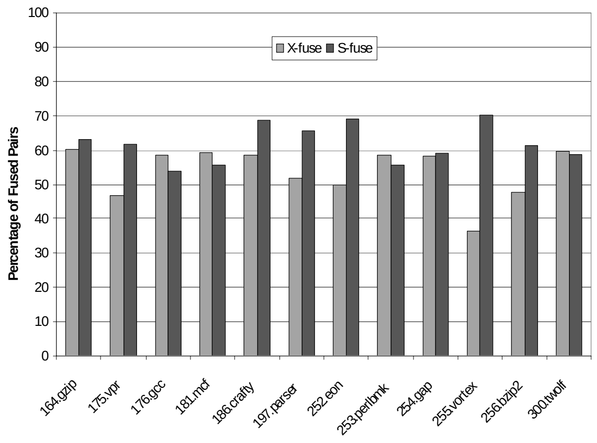


Figure 11 Fused Instruction Profile

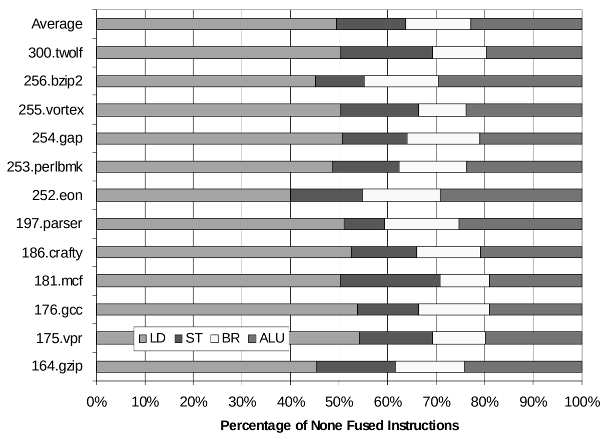


Figure 12 Non-fused Instruction Profile

5.6. Numbers of Source Operands

Finally, we consider the matter of the number of register specifiers in a scheduling window slot. As was noted earlier, in theory, pairing instructions together may require scheduling slots with twice as many source register specifiers than a conventional scheduling window slot. However, Figure 13 shows that almost all fused instruction pairs have three or fewer input register specifiers. About 95% of them have two or fewer input register specifiers. Thus, instruction fusing does not necessarily complicate the design of the instruction-scheduling window; a window slot with two source register specifiers is probably sufficient.

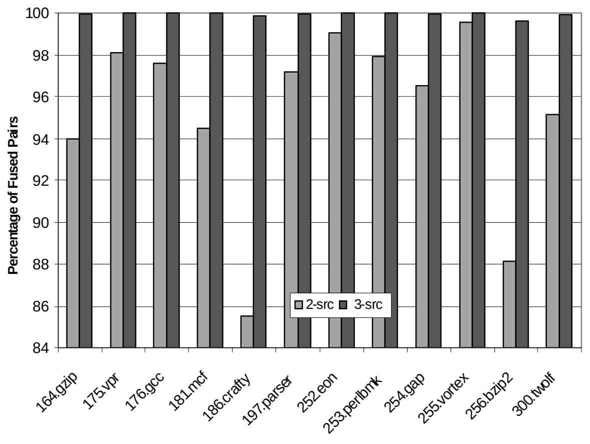


Figure 13 Number of Source Operands for Fused pairs

6. Conclusions and Future Research

The results presented here indicate that using dynamic binary translation is a good way to perform instruction fusing, especially for implementing the x86 instruction set. With regard to the fused instruction set, we conclude that (1) although there is some code expansion, we feel that it is reasonable, and is significantly less than if translation were to a conventional fixed-length RISC ISA. (2) There is a high degree of fusing, which allows each scheduling slot to handle an average of nearly 1.5 RISC operations (compared with one RISC operation in a conventional design that uses hardware cracking). (3) There are relatively few single cycle ALU instructions that are not fused, which indicates that pipelining the scheduling logic will not be especially detrimental to performance (i.e. instructions per cycle). And, (4) the individual scheduling window slots do not require more than two source register specifiers (although a third specifier will help in a small percentage of cases). Our experiments also verified the effectiveness of the algorithms developed for the dynamic binary translator. We found that they are both simple (fast) and effective.

With the binary translator in hand, we plan to complete the study of the proposed co-designed VM implementation of the x86 instruction set. The design of the microarchitecture will be completed, and a simulator will be coupled with the binary translator so that the complete VM system can be evaluated. The objective is to verify that the performance potential we have identified in this paper can, in fact, be achieved.

Acknowledgements

We thank Ilhyun Kim, Ho-Seop Kim and Jason Cantin for many helpful discussions and feedback. Comments from anonymous reviewers were also very much appreciated. This work is being supported by SRC grant 2001-HJ-902, NSF grants EIA-0071924 and CCR-0311361, Intel and IBM.

References

1. Jared Stark, Mary Brown and Yale Patt, "On Pipelining Dynamic Instruction Scheduling Logic", *Proc. of the 33rd Int'l Symp. on Microarchitecture*, pp. 57-66, Dec. 2000.

2. Mary D. Brown, Jared Stark, and Yale N. Patt, "Select-Free Instruction Scheduling logic", *Proc. of the 34th Int'l Symp. on Microarchitecture,* pp. 204-213, Dec. 2001.

3. Ilhyun Kim and Mikko H. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints", *Proc. of the 36th Int'l Symp. on Microarchitecture*, pp. 277-288, Dec. 2003.

4. Ho-Seop Kim and James E. Smith, "An Instruction Set and Microarchitecture for Instruction Level Distributed Processing", *Proc. of the 29th Int'l Symp. on Computer Architecture*, pp. 71-82, May 2002.

5. Michael Bekerman, *et al*., "Early Load Address Resolution via Register Tracking", *Proc. of the 27th Int'l Symp. of Computer Architecture*, pp. 306-315, May 2000.

6. D. Ernst and T. M. Austin, "Efficient Dynamic Scheduling Through Tag Elimination", *Proc. of the 29th Int'l Symp. on Computer Architecture*, pp. 37-46, May 2002.

7. Ilhyun Kim & Mikko H. Lipasti, "Half Price Architecture", *Proc. of the 30th Int'l Symp. on Computer Architecture,* pp. 28-38, June 2003.

8. Ramon Canal and Antonio Gonzalez, "A Low-complexity Issue Logic", *Proc. of the 14th Int'l Conf. on Supercomputing*, pp 327-335, June 2000.

9. Ramon Canal and Antonio Gonzalez, "Reducing the Complexity of the Issue Logic", *Proc. of the 15th Int'l Conf. on Supercomputing,* pp. 312-320, June 2001.

10. Pierre Michaud, André Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors", *Proc. of the 7th Int'l Symp. on High Performance Computer Architecture,* pp. 27-36, Jan. 2001.

11. L. Gwennap, "Intel P6 Uses Decoupled Superscalar Design", *Micro processor Report*, Vol. 9 No. 2, Feb. 1995.

12. Simcha Gochamn et al., "The Intel Pentium M Processor: Microarchitecture and Performance", *Intel Technology Journal,* vol7, issue 2, 2003.

13. Glenn Hinton et al., "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal*. Q1, 2001.

14. C. N. Keltcher, et al., "The AMD Opteron Processor for Multiprocessor Servers" *IEEE MICRO*, Mar.-Apr. 2003, pp. 66 -76.

15. J. M. Tendler, et al., "POWER 4 System Microarchitecture", *IBM Journal of Research and Development,* Vol. 46. No. 1, 2002.

16. Unpublished document, "CRAY-2 Central Processor", circa 1979, http://www.ece.wisc.edu/~jes/papers/cray2a.pdf

17. A. Klaiber, "The Technology Behind Crusoe Processors", Transmeta Technical Brief, 2000.

18. K. Ebcioglu et al., "Dynamic Binary Translation and Optimization", *IEEE Transactions on Computers,* Vol. 50, No. 6, pp. 529-548. June 2001.

19. Ho-Seop Kim and James E. Smith, "Dynamic Binary Translation for Accumulator-Oriented Architectures", *Proc. of the 1st Int'l Symp. on Code Generation and Optomization,* pp25-35, Mar. 2003.

20. Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *Int'l Symp. on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

21. "BOCHS: The open source IA-32 Emulation Project", http://bochs.sourceforge.net

22. Bich C. Le, "An Out-of-Order Execution Technique for Runtime Binary Translators", *Proc. of the 8th Int'l Symp. on Architecture Support for Programming Languages and Operating System",* pp. 151-158, Oct. 1998.

23. Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, 7(1-2) pp. 229-248, 1993.

24. S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock Collapsing ALUs", *IEEE Transactions on Computers*, July 1993, pp. 825-839.