An Approach for Implementing Efficient Superscalar CISC Processors

Shiliang Hu [†]   Ilhyun Kim [‡ *]   Mikko H. Lipasti [‡]   James E. Smith [‡]

shiliang@cs.wisc.edu, ilhyun.kim@intel.com, mikko@engr.wisc.edu, jes@ece.wisc.edu

Departments of [†]Computer Sciences & [‡]Electrical and Computer Engineering
University of Wisconsin - Madison

Abstract

An integrated, hardware / software co-designed CISC processor is proposed and analyzed. The objectives are high performance and reduced complexity. Although the x86 ISA is targeted, the overall approach is applicable to other CISC ISAs. To provide high performance on frequently executed code sequences, fully transparent dynamic translation software decomposes CISC superblocks into RISC-style micro-ops. Then, pairs of dependent micro-ops are reordered and fused into macro-ops held in a large, concealed code cache. The macro-ops are fetched from the code cache and processed throughout the pipeline as single units. Consequently, instruction level communication and management are reduced, and processor resources such as the issue buffer and register file ports are better utilized. Moreover, fused instructions lead naturally to pipelined instruction scheduling (issue) logic, and collapsed 3-1 ALUs can be used, resulting in much simplified result forwarding logic. Steady state performance is evaluated for the SPEC2000 benchmarks,, and a proposed x86 implementation with complexity similar to a two-wide superscalar processor is shown to provide performance (instructions per cycle) that is equivalent to a conventional four-wide superscalar processor.

1. Introduction

The most widely used ISA for general purpose computing is a CISC – the x86. It is used in portable, desktop, and server systems. Furthermore, it is likely to be the dominant ISA for the next decade, probably longer. There are many challenging issues in implementing a CISC ISA such as the x86, however. These include the implementation of complex, multi-operation instructions, implicitly set condition codes, and the trap architecture.

A major issue with implementing the x86 (and CISC ISAs in general) is suboptimal internal code sequences. Even if the original x86 binary is optimized, the many micro-ops produced by decomposing (*"cracking"*) the CISC instructions are not optimized [39]. Furthermore,

---

[*] Currently: *Intel Corporation*, Hillsboro, OR

performing runtime optimization of the micro-ops is non-trivial. In this paper, we propose and study an overall paradigm for the efficient and high performance implementation of an x86 processor. The design employs a special implementation instruction set based on micro-ops, a simplified but enhanced superscalar microarchitecture, and a layer of concealed dynamic binary translation software that is co-designed with the hardware.

A major optimization performed by the co-designed software is the combination of *dependent* micro-op pairs into fused "macro-ops" that are managed throughout the pipeline as single entities. Although a CISC ISA already has instructions that are essentially fused micro-ops, higher efficiency and performance can be achieved by first cracking the CISC instructions and then re-arranging and fusing them into different combinations than in the original code. The fused pairs increase effective instruction level parallelism (ILP) for a given issue width and reduce inter-instruction communication. For example, collapsed 3-1 ALUs can be employed to reduce the size of the result forwarding network dramatically.

Because implementing high quality optimizations such as macro-op fusing is a relatively complex task, we rely on dynamic translation software that is concealed from all conventional software. In fact, the translation software becomes part of the processor design; collectively the hardware and software become a *co-designed virtual machine* (VM) [10, 11, 29, 42] implementing the x86 ISA.

We consider an overall x86 implementation, and make a number of contributions; three of the more important are the following.

1) The co-designed VM approach is applied to an enhanced out-of-order superscalar implementation of a CISC ISA, the x86 ISA in particular.

2) The macro-op execution pipeline combines collapsed 3-1 ALU functional units with a pipelined 2-cycle macro-op scheduler. This execution engine achieves high performance, while also significantly reducing the pipeline backend complexity; for example, in the result forwarding network.

3) The advanced macro-op fusing algorithm both prioritizes critical dependences and ALU ops, and also fuses more dynamic instructions (55+ %) than reported in other work [5, 8, 27, 38] (40% or less on average) for the common SPEC2000int benchmarks.

The paper is organized as follows. Section 2 discusses related work. The co-designed x86 processor is outlined in section 3 from an architectural perspective. Section 4 elaborates key microarchitecture details. Section 5 presents design evaluation and analysis. Section 6 concludes the paper.

2. Related Work

2.1 x86 Processor Implementations

Decoder logic in a typical high performance x86 implementation decomposes instructions into one or more RISC-like micro-ops. Some recent x86 implementations have gone in the direction of more complex internal operations in certain pipeline stages, however. The AMD K7/K8 microarchitecture [9, 23] maps x86 instructions to internal Macro-Operations that are designed to reduce the dynamic operation count in the pipeline front-end. The front-end pipeline of the Intel Pentium M microarchitecture [16] fuses ALU operations with memory stores, and memory loads with ALU operations as specified in the original x86 instructions. However, the operations in each pair are still individually scheduled and executed in the pipeline backend.

The fundamental difference between our fused macro-ops and the AMD/Intel coarse-grain internal operations is that our macro-ops combine pairs of operations that (1) are suitable for processing as single entities for the entire pipeline, and (2) can be taken from different x86 instructions -- as our data shows, 70+% of the fused macro-ops combine operations from different x86 instructions. In contrast, AMD K7/K8 and Intel Pentium M group only micro-operations already contained in a single x86 instruction. In a sense, one could argue that rather than "fusing", these implementations actually employ "reduced splitting." In addition, these existing x86 implementations maintain fused operations for only part of the pipeline, e.g. individual micro-operations are scheduled separately by single-cycle issue logic.

2.2 Macro-op Execution

The proposed microarchitecture evolved from prior work on coarse-grained instruction scheduling and execution [27, 28] and a dynamic binary translation approach for fusing dependent instruction pairs [20]. The work on coarse-grained scheduling [27] proposed hardware-based grouping of pairs of dependent RISC (Alpha) instructions into macro-ops to achieve pipelined instruction scheduling. Compared with the hardware approach in [27, 28], we remove considerable complexity from the hardware

and enable more sophisticated fusing heuristics, resulting in a larger number of fused macro-ops. Furthermore, we propose a new microarchitecture in which the front-end features dual-mode x86 decoders and the backend execution engine uniquely couples collapsed 3-1 ALUs with a 2-cycle pipelined macro-op scheduler and simplified operand forwarding network. The software fusing algorithm presented here is more advanced than that reported in [20]; it is based on the observations that it is easier to determine dependence criticality of ALU-ops, and fused ALU-ops better match the capabilities of a collapsed ALU. Finally, a major contribution over prior work is that we extend macro-op processing to the *entire* processor pipeline, realizing 4-wide superscalar performance with a 2-wide macro-op pipeline.

There are a number of related research projects. Instruction-level distributed processing (ILDP)[25] carries the principle of combining dependent operations (strands) further than instruction pairs. However, instructions are not fused, and the highly clustered microarchitecture is considerably different from the one proposed here. Dynamic Strands [38] uses intensive hardware to form strands and involves major changes to superscalar pipeline stages, e.g. issue queue slots need more register tags for potentially *(n+1)* source registers of an *n-*ops strand. The Dataflow Mini-Graph [5] collapses multiple instructions in a small dataflow graph and evaluates performance with Alpha binaries. However, this approach needs static compiler support. Such a static approach is very difficult for x86 binaries because variable length instructions and embedded data lead to extremely complex code "discovery" problems [19]. CCA, proposed in [8] either needs a very complex hardware fill unit to discover instruction groups or needs to generate new binaries, and thus will have difficulties in maintaining x86 binary compatibility. The fill unit in [15] also collapses some instruction patterns. Continuous Optimization [12] and RENO [34] present novel dynamic optimizations at the rename stage. By completely removing some dynamic instructions (also performed in [39] by a hardware-based frame optimizer), they achieve some of the performance effects as fused macro-ops. Some of their optimizations are compatible with macro-op fusing. PARROT [1] is a hardware-based x86 dynamic optimization system capable of various optimizations. Compared with these hardware-intensive optimizing schemes, our software-based solution reduces hardware complexity and provides more flexibility for optimizations and implementation of subtle compatibility issues, especially involving traps [30].

2.3 Co-designed Virtual Machines

The Transmeta Crusoe and Efficeon processors [29, 42] and IBM DAISY and BOA [10, 11] are examples of co-designed VMs. They contain translation/optimization software and a *code cache*, which resides in a region of

physical memory that is completely hidden from all conventional software. In effect, the code cache [10, 11, 29] is a very large trace cache. The software is implementation-specific and is developed along with the hardware design. The co-designed VM systems from Transmeta and IBM use in-order VLIW hardware engines. As such, considerably heavier software optimization is required for translation and reordering instructions than our superscalar implementation, which is capable of dynamic instruction scheduling and dataflow graph collapsing.

3. Processor Overview

There are two major components in a co-designed VM implementation -- the *software binary translator/optimizer* and the supporting *hardware microarchitecture*. The interface between the two is the x86-specific *implementation instruction set*.

A new feature of the proposed architecture, targeted specifically at CISC ISAs, is a two-level decoder, similar in some respects to the microcode engine in the Motorola 68000 [40]. In the proposed implementation (Figure 1), the decoder first translates x86 instructions into "vertical" micro-ops -- the same fixed-format micro-ops we use as the implementation ISA (refer to the next subsection). Then, a second level decoder generates the decoded "horizontal" control signals used by the pipeline. A two-level decoder is especially suited to the x86 ISA because complex x86 instructions need to be cracked into micro-ops and then decoded into pipeline control signals. Compared with a single-level monolithic decode control table, the two-level decoder is smaller [40] by breaking the single monolithic decode table into two much smaller decode tables. It also yields decode logic that is not only more regular and flexible, but also more amenable to a fast clock.

With the two-level decoder, the pipeline can process both x86 instructions (x86-mode) and fused macro-ops (macro-op mode). When in x86-mode, instructions pass through both decode levels; this would be done when a program starts up, for example. In x86-mode, performance will be similar to a conventional x86 implementation (there is no dynamic optimization). Profiling hardware such as that proposed by Merten et al. [32] detects frequently-used code regions ("hotspots"). As hotspots are discovered, control is transferred to the VM software which organizes them into superblocks [21], translates and optimizes them as fused macro-ops, and places them in the concealed code cache. When executing this hotspot code in macro-op mode, the first level of decode in Figure 1 is bypassed, and only the second (horizontal) decode level is used. Then, the full benefits of the fused instruction set are realized. While optimized instructions are executed from the code cache, the first-level decode logic can be powered off.
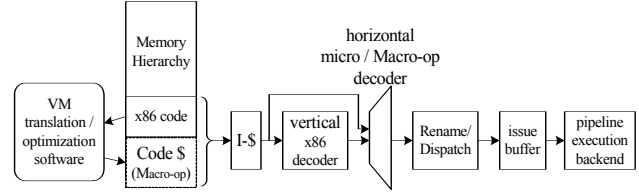

Figure 1. Overview of the proposed x86 design

As the processor runs, it will switch back and forth between x86 mode and macro-op mode, under the control of co-designed VM software. In this paper, *we focus on the macro-op mode of execution*; our goal is to demonstrate the steady-state performance benefits of the proposed x86 design. The x86 mode is intended to provide very good startup performance to address program startup concerns regarding dynamic translation. The full dual mode implementation and performance tradeoffs are the subject of research currently underway; this is being done in conjunction with our migration to a 64-bit x86 research infrastructure.

3.1 The Implementation ISA

Figure 2. Formats for fusible micro-ops

The implementation instruction set (the fusible ISA) is shown in Figure 2 and contains RISC-style micro-ops that target the x86 instruction set.

The fusible micro-ops are encoded in 16-bit and 32-bit formats. Using a 16/32-bit instruction format is not essential, but provides a denser encoding of translated instructions (and better I-cache performance) than a 32-bit only format as in most RISCs (the Cray Research and CDC machines [4, 36, 41] are notable exceptions). The 32-bit formats encode three register operands and/or an immediate value. The 16-bit formats use an x86-like 2-operand encoding in which one of the operands is both a source and a destination. This ISA is extended from an earlier version [20] by supporting 5-bit register designators in the 16-bit formats. This is done in anticipation of implementing the 64-bit x86 ISA, although results presented here are for the 32-bit version.

The first bit of each micro-op indicates whether it should be fused with the immediately following micro-op to form a single macro-op. The *head* of a fused macro-op is the first micro-op in the pair, and the *tail* is the second, dependent micro-op which consumes the value produced by the head. To reduce pipeline complexity, e.g., in the rename and scheduling stages, fusing is performed only

for dependent micro-op pairs that have a combined total of two or fewer unique input register operands. This assures that the fused macro-ops can be easily handled by conventional instruction rename/issue logic and an execution engine with a collapsed 3-1 ALU.

3.2 Dynamic Binary Translator

The major task of the co-designed dynamic binary translation software is to translate and optimize hotspot x86 instructions via macro-op fusing. Clearly, as exemplified by existing designs, finding x86 instruction boundaries and then cracking individual x86 instructions into micro-ops is lightweight enough that it can be performed with hardware alone. However, our software translation algorithm not only translates, but also finds critical micro-op pairs for fusing and potentially performs other dynamic optimizations. This requires an overall analysis of the micro-ops, reordering of micro-ops, and fusing of pairs of operation taken from different x86 instructions.

Many other runtime optimizations could also be performed by the dynamic translation software, e.g. performing common sub-expression elimination and the Pentium M's "stack engine" [16] cost-effectively in software, or even conducting "SIMDification" [1] to exploit SIMD functional units. However, in this work we do not perform such optimizations.

3.3 Microarchitecture

The co-designed microarchitecture has the same basic stages as a conventional x86 superscalar pipeline. Consequently, it inherits most of the proven benefits of such designs. A key difference is that the proposed microarchitecture can process instructions at the coarser macro-op granularity throughout the entire pipeline.

Because of the two-level decoders, there are two slightly different pipeline flows – one for executing x86 code and the other for executing optimized, macro-op code (see Figure 3). For x86 code, the pipeline operates just as a conventional dynamic superscalar processor except that the instruction scheduler is pipelined for a faster clock cycle. After the decode stage, some adjacent micro-ops cracked from x86 instructions are re-fused as in some current x86 implementations, but no reordering or optimizations are done. Note that even without optimized fusing of macro-ops, the pipeline is still a high performance superscalar processor for x86 instructions.
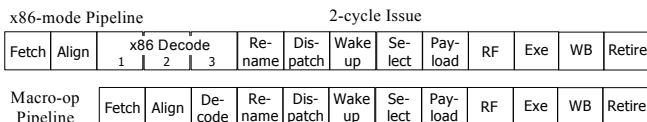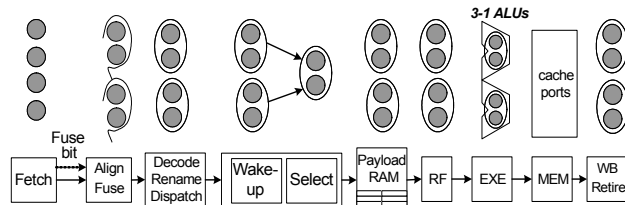

Figure 4. Macro-op execution overview

For the optimized macro-op code, paired dependent micro-ops are placed in adjacent memory locations in the code cache and are identified via a special "fuse" bit. After they are fetched the two fused micro-ops are immediately aligned and fused. From then on, macro-ops are processed throughout the pipeline as single units (Figure 4). Macro-ops contain dependent micro-ops at a granularity comparable to the original x86 CISC instructions; however, fused macro-ops are streamlined and appear as RISC-like operations to the pipeline. By processing fused micro-op pairs as a unit, processor resources such as register ports and instruction dispatch/tracking logic are reduced or better utilized. Perhaps more importantly, the dependent micro-ops in a fused pair share a single issue queue slot and are awakened and selected for issue as a single entity. The number of issue buffer slots and issue width can then be reduced without affecting performance.

After fusing, there are very few isolated single-cycle micro-ops that generate register results. Consequently, key pipeline stages can be designed as if the minimum instruction latency is two cycles. The instruction issue stage is one of the more difficult pipeline stages in a conventional design, because of the need to execute single cycle back-to-back instructions. In the proposed x86 processor design, instruction issue can be pipelined in two stages, simply and without performance loss.

Another critical stage in a conventional design is the ALU and result forwarding logic. In our design, these two operations can be performed in two cycles. In the first cycle, two dependent ALU micro-ops in a macro-op are executed by using a combination of a collapsed three-input ALU [31, 35, 37] and a conventional two-input ALU. There is no need for an expensive and time-consuming ALU-to-ALU operand forwarding network during the same cycle. Rather, the results only need to be sent to the register file (or ROB) at the end of the ALU execution cycle, and register file (or ROB) hardware can take care of providing results to dependent instructions during the next cycle as in a conventional design.

The co-designed VM CISC implementation has other advantages. For example, unused legacy features in the architected ISA can be largely (or entirely) emulated by software. A simple microarchitecture reduces design risks and cost and yields a shorter time-to-market. Although it is true that the translation software must be validated for correctness, this translation software does not require

Figure 3. x86-mode and macro-op mode pipelines

physical design checking, does not require circuit timing verification, and if a bug is discovered late in the design process, it does not require re-spinning the silicon.

4. Microarchitecture Details

The major features to support our efficient x86 processor are the software runtime macro-op fusing algorithm, and macro-op processing in the co-designed superscalar pipeline. We elaborate on technical details regarding hotspot x86 code optimization and generated macro-op code execution.

4.1 The Dynamic Translator: Macro-op Fusing

Once a hot superblock is detected, the dynamic binary translator performs translation and fusing steps. We use registers R0-R15 to map the x86 state (R0-R7 for 32-bit code), registers R16- R31 are used mainly for temporary/scratch values, x86 hotspot optimization, code cache management, precise state recovery, etc. The fusing algorithm substantially improves on the algorithm in [20]; the critical improvements will be summarized following the description of the algorithm.

Two main heuristics are used for fusing. (1) Single-cycle micro-ops are given higher priority as the head of a pair. It is easier to determine dependence criticality among ALU-ops. Furthermore, a non-fused multi-cycle micro-op will cause no IPC loss due to pipelined scheduling logic, so there is reduced value in prioritizing it. (2) Higher priority is given to pairing micro-ops that are close together in the original x86 code sequence. The rationale is that these pairs are more likely to be on the program's critical path and should be scheduled for fused execution in order to reduce the critical path latency. Consecutive (or close) pairs also tend to be less problematic with regard to other issues, e.g., extending register live ranges to provide precise x86 state recovery [30] when there is a trap. An additional constraint is maintaining the original ordering of all memory operations. This avoids complicating memory ordering hardware (beyond that used in a conventional superscalar design).

A *forward two-pass scan* algorithm creates fused macro-ops quickly and effectively (Figure 5). After constructing the data dependence graph, the first forward scan considers single-cycle micro-ops one-by-one as tail candidates. For each tail candidate, the algorithm looks backwards in the micro-op stream to find a head. This is done by scanning from the second micro-op to the last in the superblock, attempting to fuse each not-yet-fused single-cycle micro-op with the nearest preceding, not-yet-fused single-cycle micro-op that produces one of its input operands. The fusing rules favor dependent pairs with condition code dependence. And the fusing tests make sure that *no* fused macro-ops can have more than two distinct source operands, break any dependence in the original code, or break memory ordering.

```
1. for(int pass = 1; pass <=2; pass++){
2.    for(each micro-op from 2nd to last) {
3.       if(micro-op already fused)continue;
4.       if (pass == 1 and micro-op multi-cycle,
            e.g. mem-ops) continue;
5.       look backward via dependence edges for
          its head candidate;
6.       if (heuristic fusing tests pass)
            mark as a new fused pair;
7.    }
8. }
```

Figure 5. Two-pass fusing algorithm

After the first scan, a second scan is performed; the second scan allows multi-cycle micro-ops as fusing candidate tails. The lines of pseudo-code specific to the two-pass fusing algorithm are highlighted in Figure 5.

```
1. lea        eax, DS:[edi + 01]
2. mov        [DS:080b8658], eax
3. movzx      ebx, SS:[ebp + ecx << 1]
4. and        eax, 0000007f
5. mov        edx, DS:[eax + esi << 0 + 0x7c]
```
(a) x86 assembly
```
1. ADD        Reax, Redi, 1
2. ST         Reax, mem[R18]
3. LDzx       Rebx, mem[Rebp + Recx << 1]
4. AND        Reax, 0000007f
5. ADD        R21,  Reax, Resi
6. LD         Redx, mem[R21 + 0x7c]
```
(b) micro-operations
```
1. ADD  R20, Redi, 1  ::AND Reax,R20, 7f
2. ST    R20, mem[R18]
3. LDzx Rebx, mem[Rebp + Recx << 1]
4. ADD  R21, Reax,Resi::LD Redx, mem[R21 + 0x7c]
```
(c) Fused macro-ops

Figure 6. Two-pass fusing algorithm example

Figure 6 illustrates fusing of dependent pairs into macro-ops. Figure 6a is a hot x86 code snippet taken from 164.gzip in SPEC2000. The translator first cracks the x86 binary into the micro-ops, as shown in Figure 6b. Reax denotes the native register to which the x86 eax register is mapped. The long immediate 080b8658 is allocated to register R18 due to its frequent usage. After building the dependence graph, the two-pass fusing algorithm looks for pairs of dependent single-cycle ALU micro-ops during the first scan. In the example, the AND and the first ADD are fused. (Fused pairs are marked with double colon, :: in Figure 6c). Reordering, as is done here, complicates precise traps because the AND overwrites the value in register eax earlier than in the original code. Register assignment resolves this issue [30]; i.e., R20 is assigned to hold the result of the first ADD, retaining the original value of eax. During the second scan, the fusing algorithm considers multi-cycle micro-ops (e.g., memory ops) as candidate tails. In this pass, the last two dependent micro-ops are fused as an ALU-head, LD-tail macro-op.

The key to fusing macro-ops is to fuse more dependent pairs on or near the critical path. The two-pass fusing algorithm fuses more single-cycle ALU pairs on the criti-

cal path than the single-pass method in [20] by observing that the criticality for ALU-ops is easier to model and that fused ALU-ops better match the collapsed ALU units. The single-pass algorithm [20] would fuse the first `ADD` aggressively with the following store, which typically would not be on the critical path. Also, using memory instructions (especially stores) as tails may sometimes slow down the wakeup of the entire pair, thus losing cycles when the head micro-op is critical for another dependent micro-op. Although this fusing algorithm improvement comes with slightly higher translation overhead and slightly fewer fused macro-ops overall, the generated code runs significantly faster with pipelined issue logic.

Fused macro-ops serve as a means for re-organizing the operations in a CISC binary to better match state-of-the-art pipelines, e.g. most x86 conditional branches are fused with the condition test instructions to dynamically form concise branches and reduce much of the x86 condition code communication. The x86 ISA also has limited general purpose registers (especially for the 32-bit x86) and the ISA is accumulator-based, i.e. one register operand is both a source and destination. The consequent dependence graphs for micro-ops tend to be narrow and deep. This leads to good opportunities for fusing and most candidate dependent pairs have no more than two distinct source registers. Additionally, micro-ops cracked from x86 code tend to have more memory operations than a typical RISC binary; fusing some memory operations can effectively improve machine bandwidth.

4.2 The Pipeline Front-End: Macro-op Formation

The front-end of the pipeline (Figure 7) is responsible for fetching, decoding instructions, and renaming source and target register identifiers. To support processing macro-ops, the front-end fuses adjacent micro-ops based on the fuse bits marked by the dynamic binary translator.

Fetch, Align and Fuse

Each cycle, the fetch stage brings in a 16-byte chunk of instruction bytes from the L1 instruction cache. After fetch, an align operation recognizes instruction boundaries. x86-mode instructions are routed directly to the first level of the dual-mode decoders.

The handling of optimized macro-op code is similar, but the complexity is lower due to dual-length 16-bit granularity micro-ops as opposed to arbitrary multi-length, byte-granularity x86 instructions. The effective fetch bandwidth, four to eight micro-ops per cycle, is a good match for the pipeline backend. Micro-ops bypass the first level of the decoders and go to the second level directly. The first bit of each micro-op, the fuse bit, indicates that it should be fused with the immediately following micro-op. When a fused pair is indicated, the two micro-ops are aligned to a single pipeline lane, and they flow through the pipeline as a single entity.
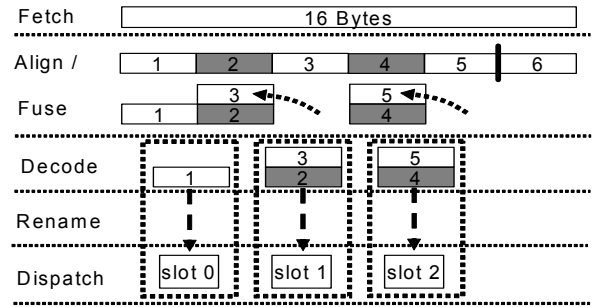

Figure 7. Front-end of macro-op execution

Instruction Decode

The x86 instructions pass through both decode levels and take three or more cycles (similar to conventional x86 processors [9, 17, 23]) for x86 cracking and decoding. RISC-style micro-ops only pass through the second level and take one cycle to decode. For each pipeline lane, decoders for micro-ops have two simple level-two micro-op decoders that can handle pairs of micro-ops (a fused macro-op pair in macro-op mode or two micro-ops in x86 mode). These micro-op decoders decode the head and tail of a macro-op pair independently of each other. Bypassing the level-one decoders results in an overall pipeline structure with fewer front-end stages when in macro-op mode than in x86 mode. The biggest performance advantage of a shorter pipeline for macro-ops is reduced branch misprediction penalties.

Rename and Macro-op Dependence Translation

Fused macro-ops do not affect register value communication. Dependence checking and map table access for renaming are performed at the individual micro-op level. Two micro-ops per lane are renamed. Using macro-ops simplifies the rename process (especially source operand renaming) because (1) the known dependence between macro-op head and tail does not require intra-group dependence checking or a map table access, and (2) there are two source operands per macro-op, which is the same for a single micro-op in a conventional pipeline.

Macro-op dependence translation converts register names into macro-op names so that issue logic can keep track of dependences in a separate macro-op level name space. In fact, the hardware structure required for this translation is identical to that required for register renaming, except that a single name is allocated to two fused micro-ops. This type of dependence translation is already required for wired-OR-style wakeup logic that specifies register dependences in terms of issue queue entry numbers rather than physical register names. This process is performed in parallel with register renaming and hence does not require an additional pipeline stage. Fused macro-ops need fewer macro-op names, thus reducing the power-intensive wakeup broadcasts in the scheduler.

Dispatch

Macro-ops check the most recent ready status of source operands and are inserted into available issue buffer and ROB entries at the dispatch stage. Because the two micro-ops in a fused pair have at most two source operands and occupy a single issue buffer slot, complexity of the dispatch unit can be significantly reduced; i.e. fewer dispatch paths are required versus a conventional design. In parallel with dispatch, the physical register identifiers, immediate values, opcodes as well as other information are stored in the payload RAM [6].

4.3 The Pipeline Back End: Macro-op Execution

The back-end of the pipeline performs out-of-order execution by scheduling and executing macro-ops as soon as their source values become available.

Instruction (Macro--op) Scheduler

The macro-op scheduler (issue logic) is pipelined and can issue back-to-back dependent *macro-ops* every two cycles. However, because each macro-op contains two dependent *micro-ops*, the net effect is the same as a conventional scheduler issuing back-to-back micro-ops every cycle. Moreover, the issue logic wakes up and selects at the macro-op granularity, so the number of wakeup tag broadcasts is reduced for energy efficiency.

Because the macro-op execution pipeline processes macro-ops throughout the *entire* pipeline, the scheduler achieves an extra benefit of higher issue bandwidth. (Macro-op execution eliminates the sequencing point at the payload RAM stage [27] that blocks the select logic for macro-op tail micro-ops).

Operand fetch: Payload RAM Access & Register File

An issued macro-op accesses the payload RAM to acquire the physical register identifiers, opcodes and other information needed for execution. Each payload RAM line has two entries for the two micro-ops fused into a macro-op. Although this configuration increases the number of bits to be accessed by a single request, the two operations in a macro-op use only a single port for both reads (the payload stage) and writes (the dispatch stage), increasing the effective bandwidth. For example, a 3-wide dispatch machine configuration has three read and three write ports that support up to six micro-ops in parallel.

A macro-op accesses the physical register file for the source values of the two fused operations. Because the maximum number of distinct source registers in a macro-op is limited to two by the dynamic binary translator, the read bandwidth is the same as for a single micro-op in a conventional implementation. Fused macro-ops better utilize register read ports by fetching an operand only once if it appears in both head and tail, and increasing the probability that both register identifiers of a macro-op are actually used. Furthermore, because we employ collapsed 3-1 ALU units at the execution stage the tail micro-op does not need the result value produced by the macro-op

head to be passed through either the register file or an operand forwarding network.

Macro-op mode does not improve register write port utilization, however, and requires the same number of write ports as a conventional machine with an equivalent number of functional units. However, macro-op execution can be extended to reduce write port requirements by analyzing the liveness of register values at translation time. We leave this to future work.
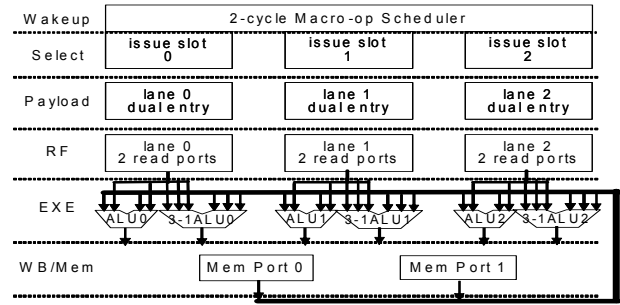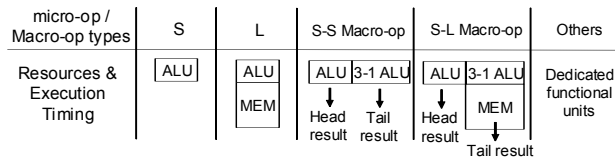

Figure 8.   Datapath for macro-op execution

Execution and Forwarding Network

Figure 8 illustrates the datapath of a 3-wide macro-op pipeline. When a macro-op reaches the execution stage, the macro-op head is executed in a normal ALU. In parallel, the source operands for both head and tail micro-ops are routed to a collapsed 3-1 ALU [31, 35, 37] to generate the tail value in a single cycle. Although it finishes execution of two dependent ALU operations in one step, a collapsed 3-1 ALU increases the number of gate levels by at most one compared with a normal 2-1 ALU [31, 35]. On the other hand, modern processors consume a significant fraction of the ALU execution cycle for operand forwarding [14, 33]. As we have already observed, the macro-op execution engine removes same-cycle ALU-ALU forwarding logic, which should more than compensate for the extra gate level for the collapsed 3-1 ALUs. Thus, the overall cycle time should not be affected by the collapsed 3-1 ALU.

To better appreciate the advantages of forwarding logic simplification, first observe that for a conventional superscalar execution engine with *n* ALUs, the ALU-to-ALU forwarding network needs to connect all *n* ALU outputs to *2*n* ALU inputs. Each forwarding path therefore needs to drive at least *2*n* loads. Typically there are other forwarding paths from other functional units such as memory ports. The implications are two-fold. (1) The many input sources at each input of the ALUs necessitate a complex MUX network and control logic. (2) The big fan-out at each ALU output means large load capacitance and wire routing that leads to long wire delays and extra power consumption. To make matters worse, when operands are extended to 64-bits, the areas and wires also increase significantly. In fact, wire issues related to forwarding led the designers of the Alpha EV6 [24] to adopt

a clustered microarchitecture. There is also a substantial body of related work (e.g. [13, 26, 33]) that attempts to address such wiring issues.

Functional units that have multiple cycle latencies, e.g. cache ports, still need a forwarding network as illustrated in Figure 8. However, the complexity of the forwarding paths for macro-op execution is much less than a conventional processor. In macro-op execution, the forwarding network only connects multi-cycle functional unit outputs to ALU inputs. In contrast, a conventional superscalar design having a full forwarding network needs to connect all input and output ports across all functional units.

micro-op / Macro-op types	S	L	S-S Macro-op	S-L Macro-op	Others
Resources & Execution Timing	ALU	ALU / MEM	ALU / 3-1 ALU → Head result / Tail result	ALU / 3-1 ALU → Head result / MEM → Tail result	Dedicated functional units

Figure 9. Resource demands and execution timing

Figure 9 illustrates resources and effective execution timings for different types of micro-ops and macro-ops; S represents a single-cycle micro-op; L represents a multi-cycle micro-op, e.g., a load, which is composed of an address generation and a cache port access. Macro-ops combining the test with the branch resolve the branch one cycle earlier than a conventional design. Macro-ops with fused address calculation ALU-ops finish address generation one cycle earlier for the LD/ST queues. These are especially effective for the x86 where complex addressing modes exist and conditional branches need separate test or compare operations to set condition codes.

Instruction Retirement

The reorder buffer performs retirement at macro-op granularity, which reduces the overhead of tracking the status of individual instructions. This retirement policy does not complicate branch misprediction recovery because a branch cannot be fused as a macro-op head. In the event of a trap, the virtual machine software is invoked to assist precise exception handling for any aggressive optimizations by reconstructing the precise x86 state (using side tables or de-optimization) [30]. Therefore, the VM runtime software enables aggressive optimization without losing intrinsic binary compatibility.

5. Evaluation

5.1 Evaluation Methodology

The proposed x86 processor is evaluated for performance via timing simulation models. The dynamic binary translator/optimizer is implemented as part of the concealed co-designed virtual machine software. The co-designed processor pipeline as described is modeled with a much modified version of SimpleScalar [7, 26] that incorporates a macro-op execution pipeline. A number of alternative x86 microarchitecture models were also simulated for comparison and performance analysis. Details regarding the microarchitecture parameters are given in Section 5.2, along with the performance results.

SPEC2000 integer benchmarks are simulated. Benchmark binaries are generated by the Intel C/C++ v7.1 compiler with SPEC2000 –O3 base optimization. Except for 253.perlbmk, which uses a small reference input data set, all benchmarks use the test input data set to reduce simulation time. All programs are simulated from start to finish. The entire benchmark suite executes more than 35 billion x86 instructions.

As stated earlier, in this work we focus on evaluating the performance of optimized macro-op code. The detailed evaluation of mixed mode x86/macro-op operation is the subject of our on-going research. Startup IPC performance in x86-mode will likely be slightly degraded with respect to conventional designs because the design is slanted toward optimized macro-op execution; results given below support this observation. The hotspot optimization overhead is negligible for most codes (including the SPEC benchmarks). With a straightforward translator / optimizer written in C++, we measure slightly more than 1000 translator instructions per single translated instruction. The SPEC benchmarks typically have hot code regions of at most a few thousand static instructions; benchmark gcc has the most hot code with almost 29,000 static instructions. The total translation overhead is thus measured in the few millions of instructions; about 30 million in the case of *gcc*. Most of the benchmarks contain over a billion instructions even for the relatively small test input set. Hence, the overhead is a fraction of one percent for all but two of the benchmarks. Overhead is largest for *gcc*, where it is two percent. With the larger reference input data, we estimate the overhead to be much smaller than one percent (.2 percent for *gcc*). This observation regarding translation overheads of under one percent is qualitatively supported by other related works in dynamic translation for SPEC2000 [3] and dynamic optimization at the x86 binary level for a set of Windows applications [32].

5.2 Performance

Pipeline models

To analyze and compare our design with conventional x86 superscalar designs, we simulated two primary microarchitecture models. The first, *baseline*, models a conventional dynamic superscalar design with single-cycle issue logic. The second model, *macro-op,* is the co-designed x86 microarchitecture we propose. Simulation results were also collected for a version of the baseline model with pipelined, two-cycle issue logic; this model is very similar to the proposed pipeline when in x86-mode. Figure 3 also serves to compare the pipeline models.

Table 1. Microarchitecture configuration

|	BASELINE	BASELINE PIPELINED	MACRO-OP
ROB Size	128	128	128
Retire width	3,4	3,4	2,3,4 MOP
Scheduler Pipeline Stages	1	2	2
Fuse RISCops ?	No	No	Yes
Issue Width	3,4	3,4	2,3,4 MOP
Issue Buffer Size	Variable. Sample points: from 16 up to 64 Effectively larger for Macro-op execution.		
Register File	128 entries, 8,10 Read ports, 5,6 Write ports		128 entries, 6,8,10 Read & Write ports
Functional Units	4,6,8 INT ALU, 2 MEM R/W ports, 2 FP ALU		
CacheHierarchy	4-way 32KB L1-I, 32KB L1-D, 8-way 1 MB L2		
Cache/Memory Latency	L1 : 2 cycles + 1 cycle AGU, L2 : 8 cycles, Mem: 200 cycles for the 1st chunk, 6 cycles b/w chunks		
Fetch width	16-Bytes x86 instructions		16B fusible RISC-ops

The *baseline* design is intended to capture the performance characteristics of a Pentium-M-like implementation although it only approximates the Pentium-M approach. First, it uses our cracked micro-ops instead of Pentium-M micro-ops (which are not available to us for obvious reasons). Second, it does not fuse the micro-ops, but has significantly wider front-end resources to provide a performance effect similar to Pentium-M micro-op fusion. In the baseline design, an "*n*-wide" baseline front-end can crack up to *n* x86 instructions per cycle, producing up to 1.5 * *n* micro-ops which are then passed up a width 1.5*n* pipeline. For example, the four-wide baseline can crack four x86 instructions into up to six micro-ops, which are then passed through a six-wide front-end pipeline. The micro-ops in the baseline are scheduled and issued separately as in current x86 processors.

Resources for the three microarchitectures are listed in Table 1. Two register read ports are reserved for stores and two write ports are reserved for loads We simulated two pipeline widths (3,4) for the baseline models and three widths (2,3,4) for the co-designed x86 processor model featuring macro-op execution.

Performance

Figure 10 shows the relative IPC performance for issue buffer sizes ranging from 16 to 64. Performance is normalized with respect to a 4-wide baseline x86 processor with a size 32 issue buffer[1]. Five bars are presented for configurations of 2-, 3-, and 4-wide *macro-op* execution model; 3- and 4-wide *baseline* superscalar.

---

[1] The normalized values are very close to the absolute values; the harmonic mean of absolute x86 IPC is 0.95 for the four-wide baseline with issue buffer size 32.
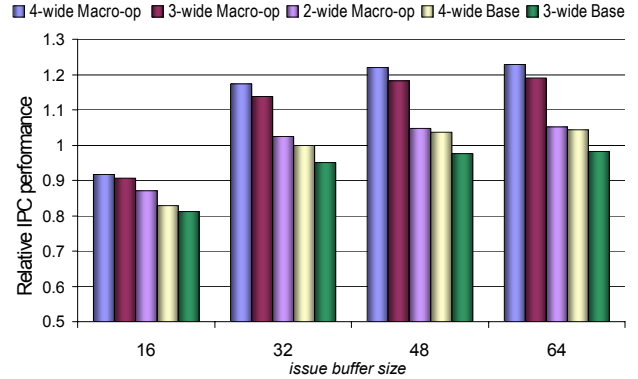
Figure 10. IPC performance comparison

If we first focus on complexity effectiveness, we observe that the two-wide co-designed x86 implementation performs at approximately the same IPC level as the four-wide baseline processor. However, the two-wide macro-op model has approximately same level of complexity as a conventional two-wide machine. The only exceptions are stages where individual micro-ops require independent parallel processing elements, i.e. ALUs. Furthermore, the co-designed x86 processor has a pipelined issue stage. Hence, we argue that the macro-op model should be able to support either a significantly higher clock frequency or a larger issue buffer for a given frequency, thus giving the same or better performance as a conventional four-wide processor. It assumes a pipeline no deeper than the baseline model, and in fact it reduces pipeline depth for hot code by removing the complex first-level x86 decoding and cracking stages from the critical branch misprediction path. On the other hand, if we pipeline the issue logic in the baseline design for a faster clock, there is an IPC performance loss of about 6~9%.

If we consider the performance data in terms of IPC alone, a four-wide co-designed x86 processor performs nearly 20% better than the baseline four-wide superscalar primarily due to its runtime binary optimization and the macro-op execution engine that has an effectively larger issue buffer and issue width. As will be illustrated in subsection 5.3, macro-op fusing increases operation granularity by 1.4. We also observe that the four-wide co-designed x86 pipeline performs no more than 4% better than the three-wide co-designed x86 pipeline and the extra complexity involved, for example, in renaming and register ports, may make the three-wide configuration more desirable for high performance.

5.3 Performance Analysis: Software Fusing

In the proposed co-designed x86 processor, the major performance-boosting feature is macro-op fusing performed by the dynamic translator. The degree of fusing, i.e., the percentage of micro-ops that are fused into pairs determines how effectively the macro-op mode can utilize the pipeline bandwidth. Furthermore, the profile of non-

fused operations implies how the pipelined scheduler may affect IPC performance. Figure 11 shows that on average, more than 56% of all dynamic micro-ops are fused into macro-ops, more than the sub-40% coverage reported in the related work [5, 8, 27, 38]. Most of the non-fused operations are loads, stores, branches, floating point, and NOPs. Non-fused single-cycle integer ALU micro-ops are only 6% of the total, thus greatly reducing the penalty due to pipelining the issue logic.

The nearly 60% fused micro-op pairs lead to an effective 30% bandwidth reduction throughout the pipeline. This number is lower than the ~65% reported in [20] because the improved fusing algorithm prioritizes critical single-cycle ALU-ops for fusing. Previous experiments with the single-pass fusing algorithm [20] actually show average IPC slowdowns because the greedy fusing algorithm does not prioritize critical dependences and single-cycle ALU operations.

Additional fusing characterization data are also collected on the SPEC 2000 integer benchmarks to evaluate the fusing algorithm and its implications on the co-designed pipeline. About 70% of the fused macro-ops are composed of micro-ops cracked from two different original x86 instructions, suggesting that inter-x86 instruction optimization is important. Among the fused macro-ops, more than 50% are composed of two single-cycle ALU micro-ops, about 18% are composed of an ALU operation head and a memory operation tail, about 30% are dynamically synthesized powerful branches, i.e. either a fused condition test with a conditional branch (mostly), or a fused ALU operation with an indirect jump in some cases.

After fusing, 46% of the fused macro-ops access two unique source registers for operands, and only 15% of fused macro-ops (about 6% among all instruction entities) write two unique destination registers. Therefore, there exist opportunities to reduce register file write ports, and further reduce pipeline complexity. Other experimental data indicate that fused micro-ops are usually close together in the micro-op sequence cracked from x86 binary.
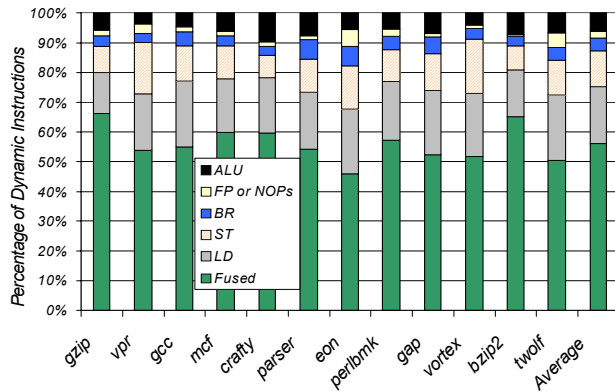
Profiling the dynamic binary translator code indicates that, because the fusing scans are not the dominant part of the translation overhead, the two-pass fusing algorithm increases binary translation overhead slightly (<10%) over a single-pass fusing algorithm.

5.4 Performance Analysis: HW Microarchitecture

In the co-designed x86 microarchitecture, a number of features all combine to improve performance. The major reasons for performance improvement are:

1) Fusing of dependent operations allows a larger effective window size and issue width, as has been noted.

2) Re-laying out code in profile-based superblocks leads to more efficient instruction delivery due to better cache locality and increased straight-line fetching. Superblocks are an indirect benefit of the co-designed VM approach. The advantages of superblocks may be somewhat offset by the code replication that occurs when superblocks are formed.

3) Fused operations lead naturally to collapsed ALUs having a single cycle latency for dependent instruction pairs. Due to pipelined (two cycle) instruction issue, the primary benefit is simplified result forwarding logic, not performance. However, there are performance advantages because the latency for resolving conditional branch outcomes and the latency of address calculation for load / store instructions is sometimes reduced by a cycle.

4) Because the macro-op mode pipeline only has to deal with RISC-like operations, the pipelined front-end is shorter due to fewer decoding stages.

Because speedups come from multiple sources, we simulated a variety of microarchitectures in order to separate the performance gains from each of the sources.

o **Baseline:** as before
o **M0:** Baseline plus superblock formation and code caching (but no translation)
o **M1:** M0 plus fused macro-ops; the pipeline length is unchanged.
o **M2:** M1 with a shortened front-end pipeline to reflect the simplified decoders for the macro-op mode.
o **Macro-op**: as before – M2 plus collapsed 3-1 ALU.

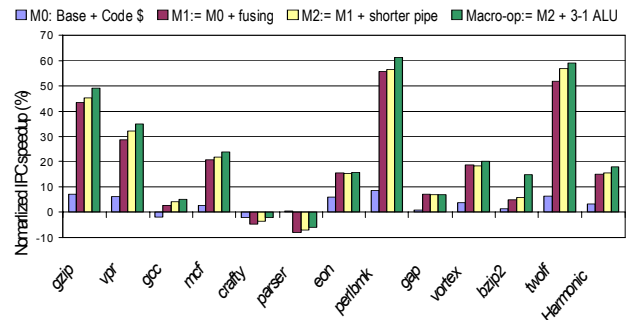Figure 11. Macro-op fusing profile

Figure 12. IPC improvement factors

All of these configurations were simulated for the four-wide co-designed x86 processor configuration featuring the macro-op execution engine, and results are normalized with respect the four-wide baseline (Figure 12). The M0 configuration shows that a hotspot code cache helps improve performance via code re-layout. The improvement is nearly 4% on average. Of course, one could get similar improvement by static feedback directed re-compilation, but this is not commonly done in practice, and with the co-designed VM approach, it happens automatically for all binaries.

The performance of M1 (when compared with M0) illustrates the gain due to macro-op fusion. This is the major contributor to IPC improvements and is more than 10% on average. The gain due to a shortened decode pipeline is nearly 1% on average. However, this gain is projected to be higher for applications where branches are less predictable. Finally, the benefit due to a collapsed ALU is about 2.5%; as noted earlier these gains are from reduced latencies for some branches and loads because the ALU result feeding these operations is sometimes available a cycle sooner than in a conventional design.

Fused macro-ops generally increase ILP by collapsing the dataflow graph. However, fusing may also cause extra cycles in some cases, e.g. when the head result feeds some other operation besides the tail, and the head is delayed because an input operand to the tail is not ready. Additionally, the pipelined scheduler may sometimes introduce an extra cycle for the 6% non-fused single-cycle ALU-ops. Figure 12 shows that our simple and fast runtime fusing heuristics may still cause slowdowns for benchmarks such as *crafty* and *parser*. The speedup for a benchmark is determined by its runtime characteristics and by how well the fusing heuristics work for it.

5.5 Discussion

Without a circuit implementation of the proposed x86 processor, some characteristics are hard to evaluate. One example is the faster clock potential that results from pipelined issue logic, removed ALU-to-ALU forwarding network, and two-level x86 decoders.

At the same pipeline width, the proposed pipeline needs more transistors for some stages, for example, ALUs, the Payload RAM and some profiling support. However, we reduce some critical implementation issues (e.g. forwarding, issue queue). Fused macro-ops reduce instruction traffic throughout the pipeline and can reduce pipeline width, leading to better complexity effectiveness and power efficiency.

6. Conclusion

Efficient and high performance x86 processors can be implemented in a co-designed virtual machine paradigm. With cost-effective hardware support and co-designed runtime software optimizers, the VM approach achieves

higher performance for macro-op mode with minimal performance loss in x86-mode (during startup). This is important for the x86 (and CISC in general) where cracking generates many micro-ops that are not optimized.

The proposed co-designed x86 processor design improves processor efficiency by reducing pipeline stage complexity for a given level of IPC performance. For complexity effective processor designs, the two-wide co-designed x86 processor significantly reduces pipeline complexity without losing IPC performance when compared with a four-wide conventional x86 superscalar pipeline. The biggest complexity savings are in the reduced pipeline width, pipelined instruction issue logic, and the removal of ALU-to-ALU forwarding paths. This reduced complexity will lead to a higher frequency clock (and higher performance), reduced power consumption, and shorter hardware design cycles.

Alternatively, with similar design complexity, the co-designed x86 macro-op execution engine improves x86 IPC performance by 20% on average over a comparable conventional superscalar design on integer benchmarks. From the IPC perspective, the largest performance gains come from macro-op fusing which treats fused micro-ops as single entities throughout the pipeline to improve ILP and reduce communication and management overhead. Our data shows that there is a high degree of macro-op fusing in typical x86 binaries after cracking, and this improves throughput for a given macro-op pipeline "width" and issue buffer size. Other features also add performance improvements of 1% to 4% each. These include superblock code re-layout (byproduct of dynamic translation), a shorter decode pipeline for optimized hotspot code, and the use of a collapsed 3-1 ALU which results in reduced latency for some branches and loads.

This study explores a CISC ISA implementation that couples the co-designed virtual machine approach with an enhanced superscalar microarchitecture. It demonstrates that this is a promising approach that addresses many of the thorny and challenging issues that are present in a CISC ISA such as the x86. The co-designed x86 processor enables efficient new microarchitecture designs while maintaining intrinsic binary compatibility.

Acknowledgements

References

1. Y. Almog *et al*. "Specialized Dynamic Optimizations for high-performance Energy-Efficient Microarchitecture", *2ⁿᵈ Int'l Symp. on Code Generation and Optimization,* 2004.

2. Vasanth Bala, *et al.* "Dynamo: A Transparent Dynamic Optimization System*", Int'l Symp. on Programming Language Design and Implementation*, pp. 1-12, 2000.

3. Leonid Baraz, *et al.* "IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems" *36th Int'l Symp. on Microarchitecture* pp 191-202 Dec. 2003.

4. P. Bonseigneur, "Description of the 7600 Computer System", *Computer Group News*, May 1969, pp. 11-15.

5. A. Bracy, P. Prahlad, A. Roth, "Dataflow Mini-Graph: Amplifying Superscalar Capacity and Bandwidth", *37th Int'l Symp. on Microarchitecture*, Dec. 2004.

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

7. D. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar ToolSet", *University of Wisconsin – Madison, Computer Sciences Department, Technical Report* CS-TR-1308, 1996.

8. N. Clark, *et al.* "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization", *37th Int'l Symp. on Microarchitecture*, 2004.

9. Keith Diefendorff "K7 Challenges Intel" *Microprocessor Report*. Vol.12, No. 14, Oct. 25, 1998

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

11. Kemal Ebcioglu, Eric R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", *24th Int'l Symp. on Computer Architecture*, 1997.

12. Brian Fahs, *et al.* "Continuous Optimization", *32nd Int'l Symp. on Computer Architecture*, 2005.

13. K. I. Farkas, *et al.* "The Multicluster Architecture: Reducing cycle time through partitioning." *30th Symp. on Microarchitecture (MICRO-30)*, Dec. 1997

14. E. Fetzer, J. Orton, "A fully bypassed 6-issue integer datapath and register file on an Itanium-2 microprocessor", *Int'l Solid State Circuits Conference*, Nov. 2002.

15. D. H. Friendly, S. J. Patel, Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors", *31st Int'l Symp. on Microarchitecture*, Dec. 1998.

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

17. L. Gwennap, "Intel P6 Uses Decoupled Superscalar Design*", Microprocessor Report*, Vol. 9 No. 2, Feb. 1995

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

19. R. N. Horspool and N. Marovac. "An Approach to the Problem of Detranslation of Computer Programs", *Computer Journal*, August, 1980.

20. Shiliang Hu and James E. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions", *2nd Int'l Symp. on Code Generation and Optimization,* March 2004.

21. Wen-Mei Hwu, *et al.* "The Superblock: An Effective Technique for VLIW and Superscalar Compilation*", The Journal of Supercomputing*, 7(1-2) pp. 229-248, 1993.

22. Quinn Jacobson and James E. Smith, "Instruction Pre-Processing in Trace Processors", *5th Int'l Symp. on High Performance Computer Architecture.* 1999

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

24. R. E. Kessler, "The Alpha 21264 Microprocessor", *IEEE Micro* Vol 19, No. 2. pp 24-36, March/April, 1999.

25. Ho-Seop Kim and James E. Smith, "An Instruction Set and Microarchitecture for Instruction Level Distributed Processing*", 29th Int'l Symp. on Computer Architecture*, 2002.

26. Ho-Seop Kim, "A Co-Designed Virtual Machine for Instruction Level Distributed Processing", Ph.D. Thesis, http://www.cs.wisc.edu/arch/uwarch/theses

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

28. Ilhyun Kim "Macro-op Scheduling & Execution*"*, Ph.D. Thesis, http://www.ece.wisc.edu/~pharm. May, 2004

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

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

31. Nadeem Malik, Richard J. Elchemeyer, Stamatis Vassilladis, "Interlock collapsing ALU for increased instruction-level parallelism", *ACM SIGMICRO Newsletter* Vol. 23, pp: 149 – 157, Dec. 1992

32. Matthew Merten, *et al.* "An Architectural Framework for Runtime Optimization". *IEEE Trans. Computers* 50(6): 567-589 (2001)

33. S. Palacharla, N. P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors", *24th Int. Symp. on Computer Architecture*, pp. 206-218, Jun, 1997

34. Vlad Petric *et al.* Tingting Sha, Amir Roth, "RENO – A Rename-based Instruction Optimizer", *32nd Int'l Symp. on Computer Architecture*, 2005.

35. J. E. Phillips, S. Vassiliadis, "Proof of correctness of high-performance 3-1 interlock collapsing ALUs", *IBM Journal of Research and Development*, Vol. 37. No. 1, 1993.

36. R. M. Russell, "The CRAY-1 Computer System" *Communications of the ACM*, Vol.21, No.1, Jan. 1978, pp.63--72.

37. Y. Sazeides, S. Vassiliadis, J. E. Smith, "The performance potential of data dependence speculation and collapsing", *29th Int'l Symp. on Microarchitecture*, 1996

38. P. Sassone, S. Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication", *37th Int'l Symp. on Microarchitecture*, Dec. 2004

39. Brian Sletchta, *et al.* "Dynamic Optimization of Micro-Operations", *9th Int'l Symp. on High Performance Computer Architecture.* Feb. 2003.

40. E. P. Stritter, H. L. Tredennick, "Microprogrammed Implementation of a Single chip Microprocessor ", *11th Annual Microprogramming Workshop*, Nov. 1978.

41. J. E. Thornton, "The Design of a Computer: the Control Data 6600", *Scott, Foresman, and Co.*, Chicago, 1970

42. Transmeta Corp. website. Transmeta Efficeon Processor.