Efficient Binary Translation

In Co-Designed Virtual Machines

by

Shiliang Hu

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2006

To my mother, and all people who have been supporting, enlightening me.

Hu, Shiliang

Abstract

There is an inherent tension between two basic aspects of computer design: standardized ISAs that allow portable (and enduring) software to be used in a wide variety of systems, and innovative ISAs that can take best advantage of ever-evolving silicon technologies. This tension originates from the ultimate objective of computer architects: efficient computer system designs that (1) support expanding capabilities and higher performance, and (2) reduce costs in both hardware and software.

This inherent tension often forces traditional processor designs out of the optimal complexity-effective envelope because a standard ISA defines the hardware/software interface and it cannot be changed without breaking binary compatibility. In this dissertation, I explore a way of transcending the limitations of conventional, standard ISAs in order to provide computer systems that are more nearly optimal in both performance and complexity. The co-designed virtual machine paradigm decouples the traditional ISA hardware/software interface. A dynamic binary translation system maps standard ISA software to an innovative, implementation-specific ISA implemented in hardware. Clearly, one major enabler for such a paradigm is an efficient dynamic binary translation system.

This dissertation approaches co-designed VMs by applying the classic approach to computer architecture: employing hardware to implement simple high performance primitives and software to provide flexibility. To provide a specific context for conducting this research, I explore a co-designed virtual machine system that implements the Intel x86 instruction set on a processor that employs the architecture innovation of macro-op execution. A macro-op is formed by fusing a dependent pair of conventional, RISC-like micro-ops.

Supported by preliminary simulation results, first I use an analytical model of major VM runtime overheads to explore an overall translation strategy. Second, I discuss efficient software binary translation algorithms that translate and fuse dependent instruction pairs into macro-ops. Third, I propose primitive hardware assists that accelerate critical part(s) of dynamic binary translation. Finally, I outline the design of a complete complexity-effective co-designed x86 processor by integrating the three major VM enabling technologies: a balanced translation strategy, efficient translation software algorithms, and simple, effective hardware primitives.

By using systematic analysis and experimental evaluations with a co-designed VM infrastructure, I reach the following conclusions.

- *Dynamic binary translation* can be modeled accurately from a memory hierarchy perspective. This modeling leads to an overall balanced translation strategy for an efficient hardware / software co-designed dynamic binary translation system that combines the capability, flexibility, and simplicity of software translation systems with the low runtime overhead of hardware translation systems.
- *Architecture innovations* are then enabled. The explored macro-op execution microarchitecture enhances superscalar processors via fused macro-ops. Macro-ops improve processor ILP as well as reduce pipeline complexity and instruction management/communication overhead.
- *The co-designed VM paradigm* is very promising for future processors. The outcomes from this research provide further evidence that a co-designed virtual machine not only provides better steady state performance (via enabling novel efficient architecture), but can also demonstrate competitive startup performance to conventional superscalar processor designs. Overall, the VM paradigm provides an efficient solution for future systems that features more capability, higher performance, and lower complexity/cost.

Acknowledgements

This dissertation research would not have been possible without the incredible academic environment at the University of Wisconsin – Madison. The education during the long six and a half years will profoundly change my life, career and perhaps more.

First, I especially thank my advisor, James E. Smith, for advising me through this co-designed x86 virtual machine research, which I enjoyed exploring during the past three or more years. It is our appreciation of the values and promises that has been motivating most of the thinking, findings and infrastructure construction work. I have learned a lot from the lucky opportunity to work with Jim and learn his approach for doing quality research, writing, thinking and evaluating things among many others.

An especially valuable experience for me was to work across two excellent departments, the Computer Sciences and the Electrical and Computer Engineering. Perhaps this was even vital for this hardware/software co-designed virtual machine research. Many research results might not have been possible without a quality background and environment in both areas. I especially appreciate the insights offered by Jim Smith, Charles Fischer, Jim Goodman, Mark Hill, Mikko Lipasti, Thomas Reps, Guri Sohi and David Wood. I remember Mark's many advices, challenges and insights during seminars and talks. I might have been doing something else if not for Mikko's architecture classes and priceless mentoring and help afterwards. I appreciate the reliable and convenient computing environment in both the departments.

The excellent Wisconsin Computer Architecture environment also manifests itself in terms of opportunities for peer learning. There are valuable discussions, peer mentoring/tutoring, reading groups, architecture lunch, architecture seminars, beers, conference travels/hanging outs and so on . I especially enjoyed and thank the companies of the Strata group. The group members are: Timothy Heil, S. S. Sastry, Ashutosh Dhodapkar, Ho-Seop Kim, Tejas Karkhanis, Jason Cantin, Kyle Nesbit, Nidhi Aggarwal and Wooseok Chang. In particular, Ho-Seop Kim shared his detailed superscalar microarchitecture timing simulator source code. Ilhyun Kim helped me to develop the microarchitecture design for my thesis research and our collaboration produced an HPCA paper. Wooseok Chang helped to setup the Windows benchmarks and trace collection tools. I learnt a lot about dissertation writing by reading other Ph.D. dissertations from Wisconsin Architecture group, especially Milo Martin's dissertation.

As a student in the CS area for more than ten years, I especially cherish the collaborations with the more than ten ECE students during those challenging ECE course projects for ECE554, 555, and 755. I learnt a lot and the experience profoundly affected my thesis research.

Prof. Chuan-Qi Zhu and BingYu Zang introduced me to computer system research and the top research teams around the world dating back to the mid-1990's, at the Parallel Processing Institute, Fudan University. I cherish the intensive mathematics training before my B.S. degree. It helped to improve the way I think and solve problems.

Finally, this research has been financially supported by the following funding sources, NSF grants CCR-0133437, CCR-0311361, CCF-0429854, EIA-0071924, SRC grant 2001-HJ-902, the Intel Corporation and the IBM Corporation. Personally, I appreciate Jim's constant and generous support. I also thank the Intel Corporation and Microsoft Research for generous multi-year scholarships and internships during my entire undergraduate and graduate career. It may not have reached this milestone without this generous support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer systems are fundamental to the infrastructure of our society. They are embodied in supercomputers, servers, desktops, laptops, and embedded systems. They power scientific / engineering research and development, communications, business operations, entertainment and a wide variety of electrical and mechanical systems ranging from aircraft to automobiles to home appliances. Clearly, the higher performance and the more capability computers can provide, the more potential applications and convenience we can benefit from. On the other hand, these computing devices often require very high hardware/software complexity. System complexity generally affects costs and reliability; more recently, it particularly affects power consumption and time-to-market. Therefore, architecture innovations that enable efficient system designs to achieve higher *performance* at lower *complexity* have always been a primary target for computer architects.

However, the several decades' history of computer architecture demonstrates that efficient designs are both application-specific and technology-dependent. In this chapter, I first discuss a dilemma that inhibits architecture innovations. Then, we outline a possible solution and the key issues to be addressed to enable such a solution. To better estimate its significance, I briefly position this thesis among the background of many related projects. Finally, we overview the thesis research and summarize the major contributions of the research.

1.1 The Dilemma: Legacy Code and Novel Architectures

Computer architects are confronted by two fundamental issues, (1) the ever-expanding and accumulating application of computer systems, and (2) the ever-evolving technologies used for implementing computing devices. A widely accepted task for computer architects is to find the optimal design point(s) for serving existing and future applications with the current hardware technology. Unfortunately, the two fundamental issues are undergoing different trends that are not in harmony with each other.

First, consider the trend for computer applications and software. We observe that for end-users or service consumers the most valuable feature of a computing device is its functional capability. Practically speaking, this capability manifests itself as the available software a computer system can run. As applications expand and accumulate, software is becoming more complex and its development, already known to be a very expensive process, is becoming more expensive. The underlying reasons are (1) computer applications themselves are becoming more complex as they expand; and (2) the conventional approach to architecture defines the hardware/software interface so that hardware implements the performance-critical primitives, and software provides the eventual solution with flexibility. Moreover, even porting a whole body of software from a binary distribution format (i.e. ISA, Instruction Set Architecture) to a new binary format is also a prohibitively daunting task. As computer applications continue to expand, a huge amount of software will accumulate. Then, it is naturally a matter of fact that software developers prefer to write code only for a standard binary distribution format to reduce overall cost. This observation about binary compatibility has been verified by the current trend in the computer industry – billions of dollars have been invested on software for the (few) surviving ISAs.

Next, turn to the other side of the architecture interface, and consider the technologies that architects rely on to implement computing devices. There has been a trend of rapidly improving and evolving technology throughout the entire history of electronic digital computers. Each technology generation provides its specific opportunities at the cost of new design challenges. It has been recognized that advanced approaches for achieving efficient designs (for a new technology generation) often require a new supporting ISA based on awareness of the technology or even dependent on the technology. For example, RISCs [103] were promoted to reduce complexity and enable single-chip pipelined processor cores. VLIW [49] was proposed as a means for further pushing the ILP envelope and reducing hardware complexity. Recently, clustered processors, for example, Multicluster [46] and TRIPS [109], were proposed for high performance, low complexity designs in the presence of wire delays [59]. Technology trends continue to present opportunities and challenges: billion-transistor chips will become commonplace, power consumption has become an acute concern, design complexity has become increasingly burdensome and perhaps even the limits of CMOS are being approached. Novel ways of achieving efficient architecture designs continue to be of critical importance.

Clearly, the two trends just described conflict with each other. On one hand, we are accumulating software for legacy ISA(s). On the other hand, in a conventional system, the ISA is the hardware/software interface that cannot be easily changed without breaking binary compatibility. Lack of binary compatibility can be fatal for some new computer designs and can severely constrain design flexibility in others. For example, RISC schemes survive more as microarchitecture designs, requiring complex hardware decoders to match legacy instruction sets such as the x86. Additionally, there is yet no evidence that VLIW can overcome compatibility issues and succeed in general-purpose computing.

Ironically, the wide-spread application of computer systems seems to be at odds with architecture innovations. And this paradox specifically manifests itself as the legacy ISA dilemma that has long been a practical reality and has inhibited modern processor designers from developing new ISA(s).

1.2 Answer: The Co-Designed Virtual Machine Paradigm

The legacy ISA dilemma results from the dual role of conventional ISA(s) as being both the software binary distribution format and the interface between software and hardware. Therefore, simply decoupling these two roles leads to a solution.

The binary format ISA used for commercial software distribution is called the *architected ISA*, for example, the x86 [6~10, 67~69] or PowerPC ISA [66]. The real interface that hardware pipeline implements, called the *implementation ISA* (or *native ISA*), is a separate ISA which can be designed with relatively more freedom to realize architecture innovations. Such innovations are keys to realize performance and/or power efficiency advantages. However, this decoupling also introduces the issue of mapping software for the architected ISA to the implementation ISA. This ISA mapping can be performed either by hardware or by software (Figure 1.1).

If the mapping is performed by hardware, then front-end hardware decoders translate legacy instructions one-by-one into implementation ISA instruction(s) that the pipeline backend can execute. For example, all recent high performance x86 processors [37, 51, 58, 74] adopt RISC microarchitecture to reduce pipeline complexity. Complex CISC decoders are employed to decompose (crack) x86 instructions into RISC-style implementation ISA instructions called *micro-ops* or *uops*. Although this context-free mapping employs relatively complex circuitry that consumes power every time an x86 instruction is fetched and decoded, the generated code is suboptimal due to inherent redundancy and

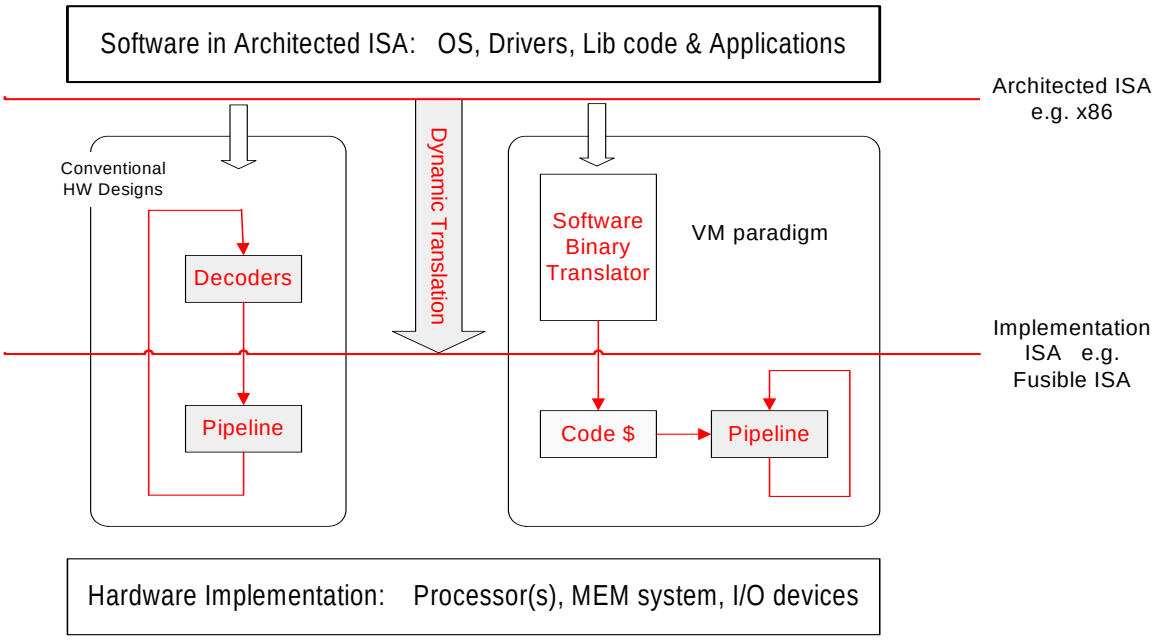


Figure 1.1 Co-designed virtual machine paradigm

inefficiency [63, 114] (Figure 1.1 left box). Therefore, as a matter of fact, to map effectively from an architected ISA to an implementation ISA, context-sensitive translation and optimization are needed to perform overall analysis over a larger translation unit, for example a basic block or a superblock [65] composed of multiple basic blocks. This kind of context-sensitive translation appears to be beyond the complexity-effective hardware design envelope.

If the mapping is performed by a concealed layer of software that is co-designed with the implementation ISA and the hardware (Figure 1.1 right box), the overall design paradigm is *a co-designed virtual machine (VM)*. The layer of concealed software is the *virtual machine monitor (VMM),* and it is capable of conducting context-sensitive ISA translation and optimization in a complexity-effective way. This VM design paradigm is exemplified in Transmeta x86 processors [82, 83], IBM DAISY [41] / BOA [3] projects and has an early variation successfully applied in IBM AS/400 systems [12, 17].

However, the co-designed VM paradigm also involves some design tradeoffs. The decoupled implementation ISA of the VM paradigm brings flexibility and freedom for realizing innovative efficient microarchitectures. But it also introduces VMM runtime software overhead for emulating the architected ISA software on the implementation ISA platform. This emulation involves dynamic binary translation and optimization that is a major source of performance overhead.

1.3 Enabling Technology: Efficient Dynamic Binary Translation

In a co-designed VM, a major component of the VMM is dynamic binary translation (DBT) that maps architected ISA binaries to implementation ISAs. And it is this ISA mapping that causes the major runtime overhead. Hence, efficient DBT is the key enabling technology for the co-designed VM paradigm.

Since a co-designed VM system is intended to enable an innovative efficient microarchitecture, it is implied that the translated native code executes more efficiently than conventional processor designs. The efficiency advantage comes from the new microarchitecture design and from the effectiveness or quality of the DBT system co-designed with the new microarchitecture. Once the architected ISA code has been translated, the processor achieves a *steady state* where it only executes native code.

Before the VM system can achieve steady state, however, the VM system first must invoke DBT for mapping ISAs, thereby incurring an overhead. This process is defined as the *startup* phase of the VM system. The translation overhead (per architected ISA instruction) of a full-blown optimizing DBT is quite heavy, on the order of thousands of native instructions per translated instruction. For example, DAISY [41] takes more than four thousands native operations to translate and optimize one PowerPC instruction for its VLIW engine. The translation (per Alpha instruction) to the supersca-

lar-like ILDP ISA takes about one thousand Alpha instructions [76, 78]. To reduce the heavy DBT overhead, VM systems typically take advantage of the fact that for most applications, only a small fraction of static instructions execute frequently (the *hotspot* code). Therefore, an adaptive/staged translation strategy can reduce overall DBT overhead. That is, staged emulation uses a light-weight interpreter or simple straightforward translator to emulate infrequent code (cold code) and thus avoid the extra optimization overhead. The reduced optimization overhead for cold code comes at the cost of inferior VM performance when emulating cold code. Both hotspot DBT optimization time overhead and inferior cold code emulation performance contribute to the so called slow startup problem for VM systems. And slow startup has long been a major concern regarding the co-designed VM paradigm because slow startup can easily offset any performance gains achieved while executing translated native code.

Figure 1.2 illustrates startup overheads using benchmarks and architectures described in more detail in Section 3.2. The figure compares startup performance of a well-tuned, state-of-the-art VM model with that of a conventional superscalar processor running a set of Windows application benchmarks. The x-axis shows time in terms of cycles on logarithmic scale. The IPC performance shown on the y-axis is normalized to steady state performance that a conventional superscalar processor can achieve. And the horizontal line across the top of the graph shows the VM steady-state IPC performance (superior to the baseline superscalar). The graphed IPC performance is the *aggregate* IPC, i.e. the total instructions executed up to that point in time divided by the total time. At a give point in time, the aggregate IPCs reflect the total numbers of instructions executed, making it easy to visualize the relative overall performance up to that time.

The relative performance curves illustrate how slowly the VM system starts up when compared with the baseline superscalar. An interesting measure of startup overhead is the time it takes for a

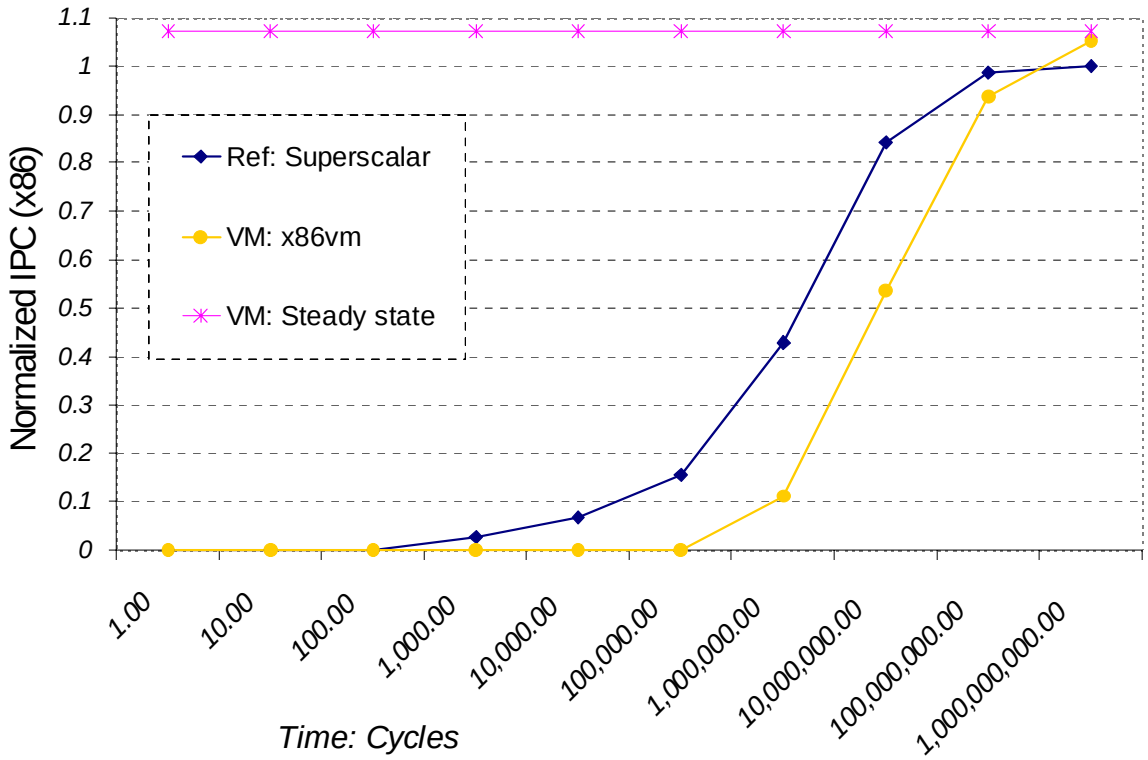


Figure 1.2 Relative performance timeline for VM components

co-designed VM to "catch up" with the baseline superscalar processor. That is, the time at which the co-designed VM has executed the same number of instructions (as opposed to the time at which the instantaneous IPCs are equal, which happens much earlier). In this example, the crossover, or breakeven, point occurs at around 200-million cycles (or 100 milliseconds for a 2.0 GHz processor core).

Clearly, long-running applications with small, stable instruction working sets can benefit from the co-designed VM paradigm with startup overheads of this magnitude. However, there are important cases where slow startup can put a co-designed VM at a disadvantage when compared with a conventional processor.

Example cases include:

- *Workloads consisting of many short-running programs or fine-grained cooperating tasks*: execution may finish before the performance lost to slow startup can be compensated for.
- *Real-time applications*: real-time constraints can be compromised if any real-time code is not translated in advance and has to go through the slow startup process.
- *Multitasking server-like systems* that run large working-set jobs: the slow startup process can be further exacerbated by frequent context switches among resource competing tasks. A limited code cache size causes hotspot translations for a switched-out task being replaced. Once the victim task is switched back in, the slow startup has to be repeated.
- *OS boot-up or shut-down*: OS boot-up/shut-down performance is important to many client side platforms such as laptops and mobile devices.

It is clear that the co-designed VM paradigm can provide a complexity-effective solution if dynamic binary translation system can be made efficient. Therefore, the major objectives of this research are to address two complementary aspects of efficient binary translation: an efficient dynamic binary translation process and efficiently executing native code generated by the translation process.

An efficient dynamic binary translation process speeds up the startup phase by reducing runtime translation overhead. Using hardware translation results in a practically zero runtime overhead at the cost of extreme complexity whereas software translation provides simplicity and flexibility at the cost of runtime overhead. Therefore, the objective here is to find hardware/software co-designed solutions that ideally demonstrate overheads (nearly) as low as purely hardware solutions, and simultaneously feature the same level of simplicity and flexibility as software solutions. The feasibil-

ity of such an overall approach relies on applying more advanced translation strategies and adding only simple hardware assists that accelerate the critical part of the translation process (again, primitives). In this thesis, we search for a comprehensive solution that combines efficient software translation algorithms, simple hardware accelerators and a new adaptive translation strategy balanced for hotspot performance advantages and its translation overhead.

Efficient native code execution affects VM performance mainly for program hotspots. The higher performance translated native code runs, the more efficiency and benefits the VM system achieves. To serve as a research vehicle that illustrates how efficient microarchitectures are enabled by the VM paradigm cost-effectively, we explore a specific co-designed x86 virtual machine in detail. This example VM features *macro-op execution* [63] to show that a co-designed virtual machine can provide elegant solutions for real world architected ISA such as the x86.

1.4 Prior Work on Co-Designed VMs

The mapping from an architected ISA to an implementation ISA is performed by either hardware or software in real processor designs.

Both Intel and AMD x86 processors [37, 51, 53, 58, 74] translate from the x86 instruction set to the internal RISC-style micro-ops (implementation ISA instructions) via hardware decoders. As already pointed out, the advantage of hardware decoders is very fast startup performance. The disadvantage is extra hardware complexity at the pipeline front-end and limited capability for translation/optimization due to context-free decoders. Regarding native code quality, it has been observed that suboptimal internal code [114] is a major issue for these hardware-intensive approaches.

Transmeta x86 processors, from the early Crusoe [54, 82] to the later Efficeon [83, 122], perform ISA mapping using dynamic binary translation systems called CMS (Code Morphing Software). These software translation systems eliminate x86 hardware decoding circuits that run continuously.

CMS exploits a staged, adaptive translation strategy to spend appropriate amount of optimizations for different parts of the program code. It performs runtime hotspot optimization cost-effectively and with more integrated intelligence. Although there is no published data about CMS runtime translation overhead, it is projected to be quite significant for benchmarks or workloads such as Windows applications [15, 82,83]. Transmeta Efficeon processors also introduced some hardware assists [83] for the CMS interpreter. However, the details are not published.

There are also prior research efforts in the co-designed VM paradigm. IBM co-designed VMs DAISY [41] BOA [3] use DBT software to map PowerPC binaries to a VLIW hardware engine. The startup performance is not explicitly addressed and the translation overhead is projected to be at least similar to that of the Transmeta CMS systems [41, 83].

A characteristic property of VM systems is that they usually feature translation/optimization software and a code cache. The code cache resides in a region of physical memory that is completely hidden from all conventional software. In effect the code cache [13, 41] is a very large trace cache. The software is implementation-specific and is developed along with the hardware design.

All the related co-designed VM systems discussed above employ in-order VLIW pipelines. As such, considerably heavier software optimization is required for translation and re-ordering instructions. In this thesis, we explore an enhanced superscalar microarchitecture, which is capable of dynamic instruction scheduling and dataflow graph collapsing for better ILP.

The ILDP project [76, 77] implements a RISC ISA (Alpha) with a co-designed VM. Because the underlying new ILDP implementation ISA and microarchitecture is superscalar-like that reorder instructions dynamically, their DBT translation is much simpler than mapping to VLIW engines. However, the startup issue was not addressed [76, 78].

This thesis explicitly addresses the startup issue, as well as the issue of quality native code generated by DBT. The approach taken in this research carries the co-designed hardware/software philosophy further by exploring simple hardware assists for DBT. The evaluation experiments are conducted for a prominent CISC architected ISA, the x86.

1.5 Overview of the Thesis Research

The major contributions in this thesis research are the following.

- ***Performance modeling of DBT systems***. A methodology for modeling and analyzing dynamic translation overhead is proposed. The new approach enables the understanding VM runtime behavior --- it models VM system performance from a memory hierarchy perspective. Major sources of overhead and potential solutions are then easily identified.

- ***Hardware / software co-designed DBT systems***. A hardware / software co-designed approach is explored for improving dynamic binary translation systems. The results support enhancing the VMM by applying a more balanced software translation strategy and by adding simple hardware assists. The enhanced DBT systems demonstrate VM startup performance that is very competitive with conventional hardware translation schemes. Meanwhile, an enhanced VM system can achieve hardware simplicity and translation/optimization capabilities similar to software translation systems.

- ***Macro-op execution microarchitecture*** (Joint work with Kim and Lipasti [63]). An enhanced superscalar pipeline, named macro-op execution, is proposed and studied to implement the x86 instruction set. The new microarchitecture shows superior steady-state performance and efficiency by first cracking x86 instructions into RISC-style micro-ops and then fusing dependent micro-op pairs into macro-ops that are streamlined for processor pipeline. Macro-ops are

treated and processed as single entities throughout the *entire* pipeline. Processor efficiency is improved because the fused dependent pairs not only reduce inter-instruction communication and instruction level management, but also collapse dataflow graph to improve ILP.

- ***An example co-designed x86 virtual machine system***. To evaluate the significance of the above individual contributions, we design an example co-designed x86 virtual machine system that features the efficient macro-op execution engine. The overall approach is to integrate the discovered valuable software strategies and hardware designs into a synergetic VM system. Compared with conventional x86 processor designs, the example VM system demonstrates overall superior steady state performance and competitive startup performance. The example VM design also inherits the complexity-effectiveness of the VM paradigm.

The rest of the dissertation is organized as follows.

Chapter 2 introduces the ***x86vm framework*** that serves as the primary vehicle for conducting this research. Then a baseline co-designed x86 virtual machine is proposed for further investigation of the VM system. The baseline VM represents a state-of-the-art VM that employs software-only DBT. Then, the three major VM components, the new microarchitecture, the co-designed VM software and the implementation ISA are described.

Chapter 3 addresses the translation ***strategy***. It presents a performance modeling methodology for VM systems from a memory hierarchy perspective. The dynamics of translation-based systems are explored within this model. Then, an overall translation strategy for reducing VM runtime overhead is proposed.

Chapter 4 addresses the translation ***software*** that determines the efficiency of translated native code in the proposed VM system. I discuss the major technical issues such as translation and optimi-

zation algorithms that generate efficient native code for the macro-op execution microarchitecture. Meanwhile, the algorithms are aware of translation efficiency to reduce overhead.

Chapter 5 addresses the translation ***hardware*** support. I propose simple hardware assists for binary translators. This chapter discusses the hardware assists from architecture, microarchitecture, and circuit perspectives, along with some analysis of their complexity. I also discuss other related hardware assists that are not explicitly studied in this thesis.

Chapter 6 emphasizes balanced ***synergetic integration*** of all VM aspects addressed in the thesis via a complete example co-designed x86 virtual machine system. The complete VM system is evaluated and analyzed with respect to the specific challenges architects are facing today or will face in the near future. Evaluations are conducted via microarchitecture timing simulation.

Chapter 7 summarizes and concludes the thesis research.

Because co-designed virtual machine systems involve many aspects of hardware and software, I evaluate individual thesis features and discuss the related work in each chapter. That is, evaluation and related work are distributed among the chapters.

Chapter 2

The *x86vm* Experimental Infrastructure

The x86 instruction set [67~69, 6~10] is the most widely used ISA for general purpose computing. The x86 is a complex instruction set that pose many challenges for high-performance, power-efficient implementations. This makes it an especially compelling target for innovative, co-designed VM implementations and underlying microarchitectures. Consequently, the x86 was chosen as the architected ISA for this thesis research.

As part of the thesis project, I developed an experimental framework named ***x86vm*** for researching co-designed x86 virtual machines. This chapter briefly introduces the *x86vm,* framework, including its objectives, high-level organization, and evaluation methodology. I use this infrastructure first to characterize x86 applications and identify key issues for implementing efficient x86 processors. The results of this characterization suggest a new efficient microarchitecture employing *macro-op execution* as the execution engine for the co-designed VM system. This microarchitecture forms the basis of the co-designed x86 virtual machine that is developed and studied in the remainder of the thesis.

2.1 The *x86vm* Framework

The co-designed VM paradigm adds flexibility and enables processor architecture innovations that may require a new ISA at the hardware/software interface. Therefore, there are two major components to be modeled in a co-designed VM experimental infrastructure. The first is the co-designed software VMM and the other is the hardware processor. The interface between the two components is the *implementation ISA*.

There are several challenges for developing such an experimental infrastructure, especially in an academic environment. The most important are: (1) The complexity of microarchitecture timing model for a co-designed processor is of the same as for a conventional processor design. (2) In a research environment, the implementation ISA is typically not fixed nor defined at the beginning of the project. (3) Dynamic binary translation is a major VMM software component. Although there are many engineering tradeoffs in implementing dynamic binary translation, for the most part experimental data regarding these tradeoffs has not been published. Moreover, because of its complexity, a dynamic binary translation system for the x86 ISA is an especially difficult one.

Figure 2.1 sketches the *x86vm* framework that I have developed to satisfy the infrastructure challenges. There are two top-level components. The *x86vmm* component models the software VMM system, and the *microarchitecture* component models the hardware implementation for the processor core, caches, memory system etc. The interface between the two is an abstract ISA definition. These top-level components and interface should be instantiated into concrete implementations for a specific VM design and evaluation. In this section, I overview high level considerations and trade-offs regarding instantiation of these top level components.

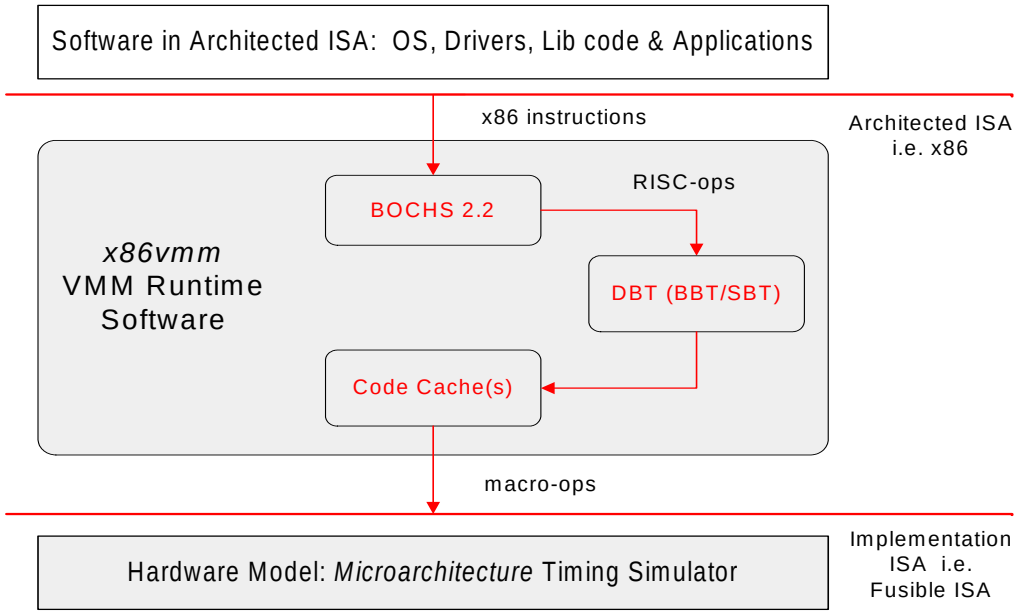


Figure 2.1 The *x86vm* Framework

The VMM components (upper shaded box in Figure 2.1) are modeled directly by developing the VM software as part of the VM design. To support modeling of a variety of x86 workloads, which employ a wide variety of the x86 instructions, I extracted the x86 decode and x86 instruction emulation semantic routines from BOCHS 2.2 (A full system x86 emulation system [84]). In each x86 instruction semantic routine, I added additional code to crack the x86 instruction into abstract RISC-style micro-ops. For a specific VM design, these abstract micro-ops are translated by the dynamic binary translation system into implementation ISA instructions that are executed on the specific co-designed processor.

The implementation ISA is one of the important research topics in this thesis. An early instantiation of the framework briefly explored an ILDP ISA [76]. The eventual implementation ISA is a (RISC-style) ISA named the *fusible instruction set*, which will be overviewed in Section 2.4.

The microarchitecture components (in the lower shaded box in Figure 2.1) are modeled via detailed timing simulators as is done in many architecture studies. For the fusible instruction set, I developed a microarchitecture simulator based on H.-S. Kim's IBM POWER4-like detailed superscalar microarchitecture simulator [76]. To address x86 specific issues, I adapted it and extended it to model the new macro-op execution microarchitecture.

The timing simulators in the *x86vm* infrastructure are trace-driven. The reason for using trace-driven is primarily to reduce the amount of engineering for developing this new infrastructure. However, there are implications due to trace-driven simulations: (1) Trace-driven simulations do not perform functional emulation simultaneously with timing evaluation. Therefore, there is no guarantee that the timing pipeline produces exactly the same results as an execution-driven simulator. In this thesis research, we inspected the translated code, and verified that the simulated instructions are the same (although re-ordered). However, there is no verification of the execution results produced by the timing pipeline as timing models do not calculate values. (2) Trace-driven timing models also lose some precision for timing/performance numbers. For example, "wrong-path" instructions are not modeled. Wrong path instructions may occasionally prefetch useful data and/or pollute the data cache. Similarly, branch predictor and instruction cache behavior may be affected. In many cases, these effects cancel each other, in others they do not.

The primary ISA emulation mechanism is dynamic binary translation (DBT), but other emulation schemes such as interpretation and static binary translation are sometimes used. In the design of DBT systems, there are many trade-offs to be considered, for example: (1) choosing between an optimizing DBT or a simple light-weight translation, (2) deciding the (number of) stages of an adaptive/staged translation system and (3) determining the transition mechanisms between the stages.

Static translation does not incur runtime overhead. However, it is very difficult, if possible at all, to find all the individual instructions in a static binary (code discovery [61]) for a variable length ISA such as the x86, which also allows mixing data with code. Additionally, for flexibility or functionality, many modern applications execute code that is dynamically generated or downloaded via a network. Static binary translation lacks the capability to support dynamic code and dynamic code optimization.

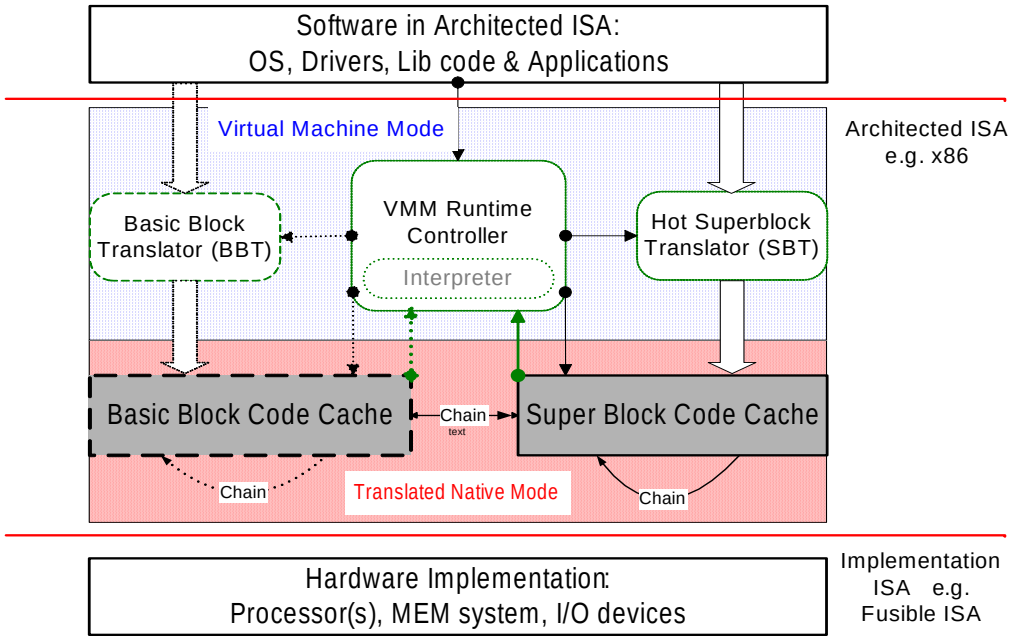
The emulation speed of an interpreter is typically 10X to 100X slower than native execution. Some VM systems employ an interpreter to avoid performing optimizations on non-hotspot code that usually occurs during the program startup phase. An alternative (sometimes an addition to) interpretation is simple basic block translation (BBT) that translates code one basic block a time without optimization. The translated code is placed in a code cache for repeated reuse. For most ISAs, the simple BBT translation is generally not much slower than interpretation, so most recent binary translation systems skip interpretation and immediately begin execution with simple BBT. The Intel IA-32 EL [15] uses this approach, for example.

For a co-designed VM, full ISA emulation is needed to maintain 100% binary compatibility with the architected ISA, and high performance emulation is necessary to unleash all the advantages of new efficient processor designs. Therefore, the *x86vm* framework adopts a DBT-only approach for ISA emulation. For complexity-effectiveness, a two-stage adaptive DBT system is modeled in the framework. This adaptive system uses a simple basic block translator (BBT) for non-hotspot code emulation and a superblock translator (SBT) for hotspot optimization. The terminology used in this thesis is that DBT is the generic term that includes both BBT and SBT as special cases. The dynamics and trade-offs behind a two-stage translation system will be further discussed in Chapter 3 where

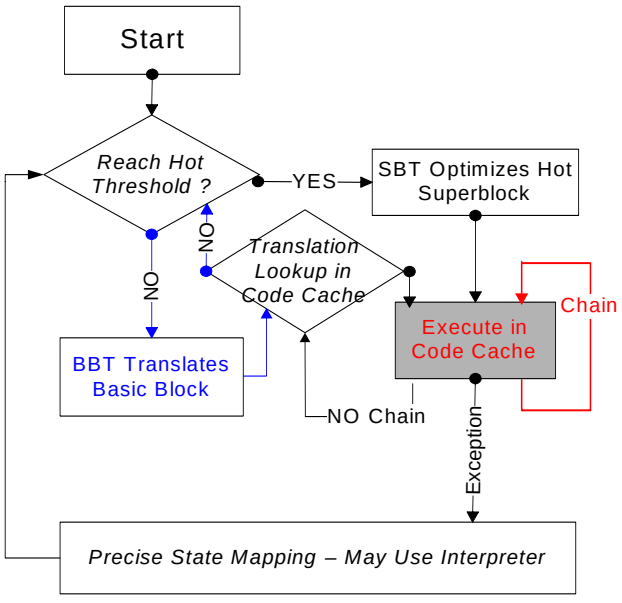
DBT performance modeling and analysis are systematically considered. In this section, we outline the high level organization of the DBT translation framework.

There are four major VMM components (Figure 2.2a) in the *x86vm* framework. (1) A simple light-weight basic block translator (*BBT*) that generates straightforward translations for each basic block when it is first executed; (2) An optimizing *superblock* binary translator (*SBT*) that optimizes hotspot superblocks; (3) Code caches – concealed VM memory areas for holding BBT and SBT translations; and (4) the *VMM runtime system* that orchestrates the VM execution: it executes translation strategy by selecting between BBT and SBT for translation; it recovers precise program state and manages the code caches, etc.

Figure 2.2b is the VM software flowchart. When an x86 binary starts execution, the system enters the VM software (*VM mode*) and uses the BBT translator to generate fusible ISA code for initial emulation (Figure 2.2b). Once a hotspot superblock is detected, it is optimized by the SBT system and placed into the code cache. Branches between translation blocks may be linked initially by the VMM runtime system via translation lookup table, but are eventually chained directly in the code cache. For most applications, the VM software will quickly find the instruction working set, optimize it, and then leave the processor executing in the translated code cache as the steady state, which is defined as the *translated native mode* (shaded in Figure 2.2).



(a)



(b)

Figure 2.2 Staged Emulation in a Co-Designed VM

For driving the trace-driven timing pipeline with the translated code blocks, the *x86vmm* run-time controller object has a special method, named *exe_translations*. Whenever the *x86vm* system has translated code for the instruction sequence from the x86 trace stream, this method verifies and ensures that the translated code correctly follows the corresponding x86 instruction stream. Then it feeds the timing model with the translated code sequence. Memory addresses from the x86 trace steam are also passed to the corresponding translated native *uops* to model the memory system correctly. The fetch stage of the timing pipeline reads the output stream of this method and models timing for fetching such as I-TLB, I-cache and branches.

2.2 Evaluation Methodology

For evaluating the proposed co-designed VM, we use the currently dominant processor design scheme, the superscalar microarchitecture, as the reference/baseline system. Ideally, the reference system would accurately model the best-performing x86 processor. However, for practical reasons (not the least of which are intellectual property issues), such a reference system is not available. For example, the internal micro-ops and key design details/trade-offs for real x86 processors are not publicly available. Consequently, the reference x86 processor design in this research is an amalgam of the AMD K7/K8 [37,74] and Intel Pentium M [51] designs. The reference design is based on machine configuration parameters such as pipeline widths, issue buffer sizes, and branch predictor table sizes that are published. The detailed reference configuration will be described in more detail in the specific evaluation sections.

Performance evaluation is conducted via detailed timing simulation. The simulation models for different processor designs are derived from the *x86vm* framework. For the reference x86 processors, modified BOCHS 2.2 [84] x86 decode/semantic routines are used for functional simulation. Then, RISC micro-ops are generated from the x86 instructions for simulation with the reference x86 timing

simulator. The reference timing model is configured to be similar to the AMD K7/K8 and Intel Pentium M designs. For the co-designed VM designs, dynamic binary translators are implemented as part of the concealed co-designed virtual machine software. A simulation model of the *x86vm* pipeline is used for accurately modeling the detailed design of the various co-designed processor cores.

The SPEC2000 integer benchmarks and Winstone2004 Business Suite are selected as the simulation workload. A brief description of the benchmarks given in Table 2.1.

Benchmark binaries for the SPEC2000 integer benchmarks 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, the SPEC2000 benchmarks use the test input data set to reduce simulation time. SPEC2000 binaries are loaded by *x86vm* into its memory space and emulated by the extracted BOCHS2.2 code to generate “dynamic traces” for the rest of the simulation infrastructure. The adapted BOCHS code can also generate uops while performing the functional simulation.

Winstone2004 is distributed in binary format with an embedded data set. Full system traces are collected randomly for all the Windows applications running on top of the Windows XP operating system. A colleague, W. Chang, installed Window XP with the SP2 patch inside SimICS [91] and set up the Winstone 2004 benchmark. This system was then used for collecting traces that serve as x86 trace input streams to the x86vm framework. When processing these x86 trace files, the *x86vm* infrastructure does not need to perform functional emulation.

Table 2.1 Benchmark Descriptions

SPEC2K INTEGER	BRIEF DESCRIPTION
164.gzip	Data compression utility
175.vpr	CAD tool : FPGA circuit placement and routing
176.gcc	C/C++ Compiler.
181.mcf	Minimum cost flow network
186.crafty	Artificial Intelligence: Chess program
197.parser	Artificial Intelligence: Natural language processing
252.eon	Computer Graphics: Ray tracing
253.perlbmk	Perl scripts execution environment.
254.gap	Computational group theory
255.vortex	Objected oriented database system.
256.bzip2	Data compression utility.
300.twolf	CAD tool for electronic design: place and route simulator.

WINSTONE2004 BENCH	BRIEF DESCRIPTION
Access	Databases, reports
Excel	Data processing spread sheet.
Front Page	Document processing application.
Internet Explorer	Internet webpage browsing application
Norton Anti-virus	Safe work environment: Anti-virus protection system.
Outlook	Emails, calendars, scheduler.
Power Point	Document, Presentation utility.
Project	Project planning, management tool
Win-zip	Data archive, compression utility.
Word	Document editing application.

There are two important performance measurements, steady state performance and the startup performance. For steady state performance evaluation, long simulations are run to ensure steady state is reached. The SPEC2000 CPU benchmarks runs are primarily targeted at measuring steady state performance. All programs in SPEC2000 are simulated from start to finish. The entire benchmark suite executes more than 35 billion x86 instructions. For startup performance measurement, short

simulations that stress startup transient behavior are used. Because Windows applications are deemed to be challenging for startup performance, especially for binary translation based systems, we focus on Windows benchmarks for the startup performance study.

2.3 x86 Instruction Characterization

The x86 instruction set uses variable-length instructions that provide good code density. ISA code density is important for both software binary distribution and high performance processor implementation. A denser instruction encoding leads to smaller code footprint that can help mitigate the increasingly acute memory wall issue and improve instruction fetch efficiency. However, good x86 code density comes at the cost of complex instruction encoding. The x86 encoding often assumes implicit register operands and combines multiple operations into a single x86 instruction. Such a complex encoding necessitates complex decoders at the pipeline front-end. We characterize x86 instructions for the SPEC2000 integer and the WinStone2004 Business workloads. The goal is to search for an efficient new microarchitecture and implementation ISA design.

Because most x86 implementations decompose, or *crack*, the x86 instructions into internal RISC style micro-ops. Many CISC irregularities such as irregular instruction formats, implicit operands and condition codes, are streamlined for a RISC core during the CISC-to-RISC cracking stage. However, cracking each instruction in isolation does not generate optimal micro-op sequences even though the CISC (x86) binaries are optimized. The “context-free” cracking will result in redundancies and inefficiencies. For example, redundant address calculations among memory access operations, redundant stack pointer updates for a sequence of x86 *push* or *pop* instructions [16], inefficient communication via condition flags due to separate branch condition tests and the corresponding branch instructions. Moreover, the cracking stage generates significantly more RISC micro-ops than x86 instructions that must be processed by the backend execution engine.

Table 2.2 CISC (x86) application characterization

	DYNAMIC INSTRUCTION COUNT EXPANSION	STATIC FIXED 32-BIT RISC CODE EXPANSION	STATIC 16 / 32 - BIT RISC CODE EXPANSION	
SPEC 2000 CPU integer				
164.gzip	1.54	1.63	1.18	
175.vpr	1.44	2.06	1.39	
176.gcc	1.34	1.81	1.32	
181.mcf	1.40	1.65	1.21	
186.crafty	1.50	1.64	1.23	
197.parser	1.42	2.08	1.42	
252.eon	1.56	2.21	1.47	
253.perlbmk	1.53	1.84	1.29	
254.gap	1.31	1.88	1.32	
255.vortex	1.50	2.11	1.41	
256.bzip2	1.46	1.79	1.33	
300.twolf	1.26	1.65	1.18	
SPEC2000 average	1.44	1.86	1.31	
WinStone2004 business suites				
Access	1.54	2.06	1.41	
Excel	1.60	2.02	1.39	
Front Page	1.62	2.29	1.52	
Internet Explorer	1.58	2.45	1.72	
Norton Anti-virus	1.39	1.57	1.20	
Outlook	1.56	1.96	1.35	
Power Point	1.22	1.58	1.18	
Project	1.67	2.35	1.56	
Win-zip	1.18	1.76	1.23	
Word	1.61	1.79	1.29	
Winstone average	1.50	1.98	1.39	

Table 2.2 lists some basic characterization of the x86 applications benchmarked. The first data column shows that on average, each x86 instruction cracks into 1.4 ~ 1.5 RISC-style micro-ops. This dynamic micro-op expansion not only stresses instruction decode/rename/issue logic (and add overhead), but also incur unnecessary inter-instruction communication among the micro-ops that stresses the wire-intensive operand bypass network.

Meanwhile, the CISC-to-RISC decoders are already complex logic because the x86 ISA tends to encode multiple operations without strict limits on instruction length. The advantage of this x86 property is concise instruction encoding and consequently a smaller code footprint. The disadvantage is the complexity that hardware decoders must handle for identifying variable-length instructions and cracking CISC instructions into RISC micro-ops. Multiple operations inside a single CISC instruction need to be isolated and reformatted for the new microarchitecture.

To be more specific, the length of x86 instructions varies from one byte to seventeen bytes. Figure 2.3 shows that 99.6+% dynamic x86 instructions are less than eight bytes long. Instructions more than eleven bytes are very rare. The average x86 instruction length is three bytes or fewer. However, the wide range of instruction lengths makes the x86 decoders much more complex than RISC decoders. For a typical x86 decoder design, the critical path of the decoder circuit is to determine boundaries among the x86 instruction bytes. Moreover, the CISC-to-RISC cracking further increases CISC decoding complexity because it needs additional decode stage(s) to decompose CISC instructions into micro-ops.

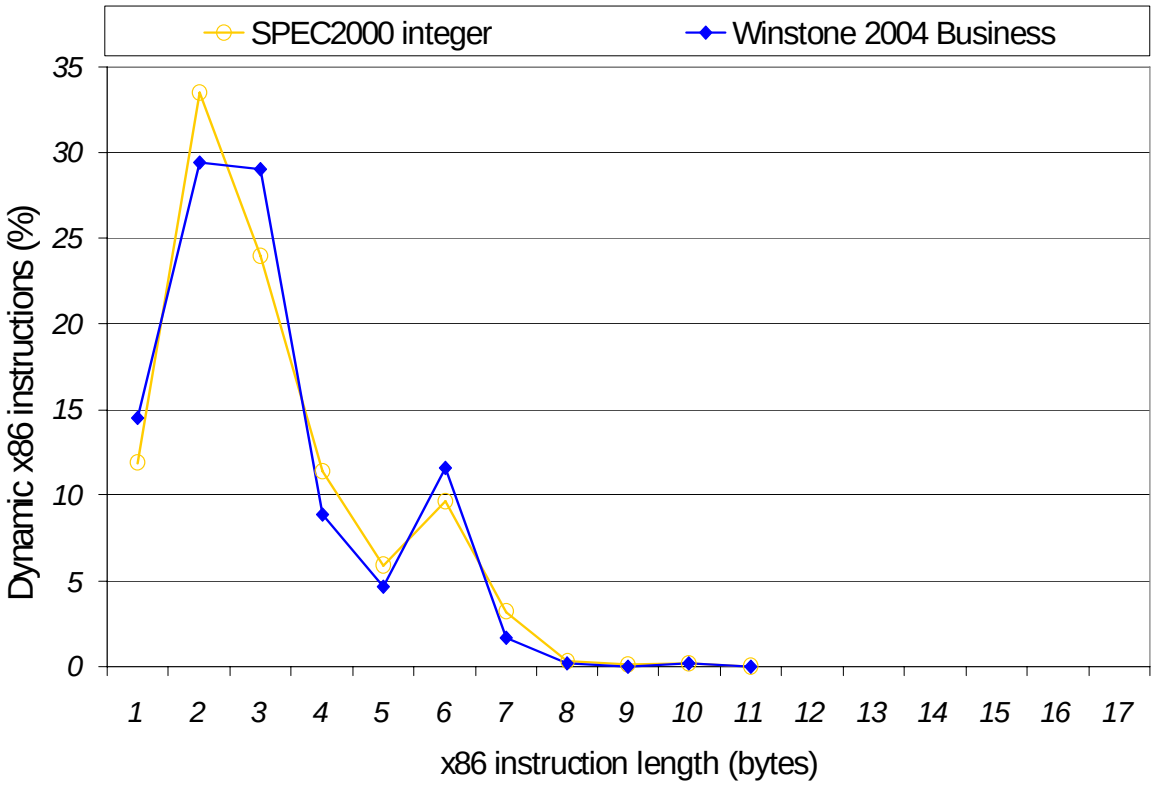


Figure 2.3 Dynamic x86 instruction length distribution

On the other hand, by combining these two factors (variable-length instructions and CISC-to-RISC cracking ratio), it is clear that the x86 code density is nearly twice as good as typical RISC ISAs. The second data column of Table 2.2 verifies this observation with benchmark characterization data. The third column of Table 2.2 illustrates that a RISC ISA can narrow this code density gap by adopting a 16/32-bit instruction encoding scheme. This limited variable length encoding ISA represents a trade-off between code density and decoder complexity that has long been implemented in early RISC designs such as the CDC and Cray Research machines [19, 32, 33, 34, 107, 121].

For a brief summary of the major CISC (x86) specific challenges, we observe that an efficient microarchitecture design needs to address the suboptimal internal micro-op code and to balance code

density with decoder complexity. Complex decoders not only complicate circuit design, but also consume power.

An additional concern regarding an architected ISA such as the x86 is the presence of "legacy" features. For the x86 instruction set [67, 68, 69] new instructions have been being added to better support graphics/multimedia and ISA virtualization, and many other features have become practically obsolete. For example, the virtual-8086 mode and the x86 BCD (binary coded decimal) instructions are rarely used in modern software. The x86 segmented memory model is largely unused and the segment registers are disabled altogether in the recent x86 64-bit mode [6~10] (Except FS and GS that are used as essentially additional memory address registers). Conventional processor designs have to handle all these legacy features of the ISA. A new efficient design should provide a solution such that obsolete features will not complicate processor design.

2.4 Overview of the Baseline *x86vm* Design

A preliminary co-designed x86 VM is developed to serve as the baseline design for investigating high performance dynamic binary translation. The two major VM components, the hardware microarchitecture and the software dynamic binary translator, are both modeled in the *x86vm* framework. As in most state-of-the-art co-designed VM systems, the baseline VM features very little hardware support for accelerating and enhancing dynamic binary translation. Further details and enhancements to the baseline VM design will be systematically discussed in the next three chapters that address different VM design aspects.

2.4.1 Fusible Implementation ISA

The internal fusible implementation ISA is essentially an enhanced RISC instruction set. The ISA has the following architected state:

- The program counter.
- 32 general-purpose registers, R0 through R31, each 64-bit wide. Reads to R31 always return a zero value and writes to R31 have no effect on the architected state.
- 32 FP/media registers, F0 through F31, each 128-bit wide. All x86 state for floating-point and multimedia extensions, MMX / SSE(1,2,3) SIMD state, can be mapped to the F registers.
- All x86 condition code and flag registers (x86 EFLAGS and FP/media status registers) are supported directly.
- System-level and special registers that are necessary for efficient x86 OS support.

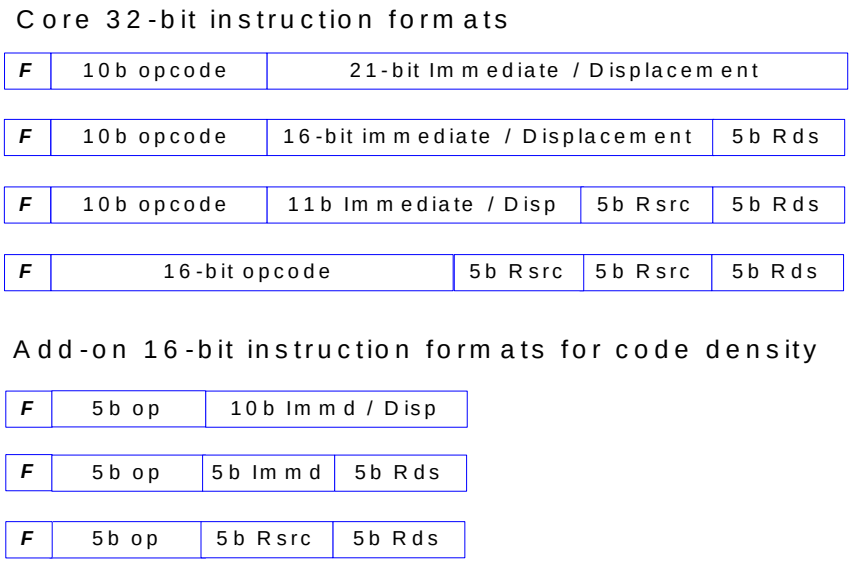


Figure 2.4 Fusible ISA instruction formats

The fusible ISA instruction formats are illustrated in Figure 2.4. The instruction set adopts RISC-style micro-ops that can support the x86 instruction set efficiently. The fusible micro-ops are encoded in either 32-bit or 16-bit formats. Using a 16/32-bit instruction format is not essential. However, as shown in Table 2.2, it provides a denser encoding of translated instructions and better instruction-fetch performance than a 32-bit only format. The 32-bit formats are the “kernel” of the ISA and encode three register operands and/or an immediate value. The 16-bit formats employ an x86-style accumulator-based 2-operand encoding in which one of the operands is both a source and a destination. This encoding is especially efficient for micro-ops that are cracked from x86 instructions. All general-purpose register designators (R and F registers) are 5-bit in the instruction set. All x86 exceptions and interrupts are mapped directly onto the fusible ISA.

A special feature of the fusible ISA is that a pair of dependent micro-ops can be fused into a single *macro-op*. The first bit of each micro-op indicates whether it should be fused with the immediately following micro-op to form a macro-op. We define the *head* of a macro-op as the first micro-op in the pair, and define the *tail* as the second, dependent micro-op which consumes the value produced by the head. To reduce pipeline complexity, e.g., in the renaming and scheduling stages, we only allow fusing of dependent micro-op pairs that have a combined total of two or fewer unique source register-operands. This ensures that the fused macro-ops can be easily handled by conventional instruction rename/issue logic and an execution engine featuring collapsed 3-1 ALU(s).

To support x86 address calculations efficiently, the fusible instruction set adopts the following addressing modes to match the important x86 addressing modes.

- Register indirect addressing: ***mem[register]***;
- Register displacement addressing: ***mem[register + 11bit_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.

The fusible ISA assumes a flat, page-based virtual memory model. This memory model is the dominant virtual memory model implemented in most modern operating system kernels, including those running on current x86 processors. Legacy segment-based virtual memory can be emulated in the fusible ISA via software if necessary.

In the instruction formats shown in Figure 2.4, 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.

2.4.2 Co-Designed VM Software: the VMM

As observed in Section 2.3, cracking x86 instructions into micro-ops in a context-free manner simplifies the implementation of many complex x86 instructions; it also leads to significantly more micro-ops to be managed by the pipeline stages. Therefore, fusing some of these micro-ops improves pipeline efficiency and performance because fused macro-ops collapse the dataflow graph for better ILP and reduce unnecessary inter-instruction communication and dynamic instruction management. Hence, the major task of our co-designed dynamic binary translation software is to translate and optimize frequently used, "hotspot" blocks of x86 instructions via macro-op fusing. Hot x86 instructions are first collected as hot superblocks and then are "cracked" into micro-ops. Dependent micro-op pairs are then located, re-ordered, and fused into macro-ops. The straightened, translated, and optimized code for a superblock is placed in a concealed, non-architected area of main memory -- the code cache.

As is evident in 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 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 groups of micro-ops, re-ordering of micro-ops, and fusing of pairs of operations taken from different x86 instructions. To keep our x86 processor design complexity-effective, we employ software translation to perform runtime hotspot optimization.

We note that the native x86 instruction set already contains what are essentially fused operations. However, our dynamic binary translator often fuses micro-op pairs across x86 instruction boundaries and in different combinations than in the original x86 code. To achieve the goal of streamlining the generated native code for the macro-op execution pipeline, our fusing algorithms

fuse pairs of operations that are not permitted by the x86 instruction set; for example the pairing of two ALU operations and the fusing of condition test instructions with conditional branches.

Although it is not done here, it is important to note that many other runtime optimizations can be performed by the dynamic translation software, e.g. performing common sub-expression elimination and the Pentium M's "stack engine" [16, 51] cost-effectively in software, or even conducting "SIMDification" [2] to exploit SIMD functional units.

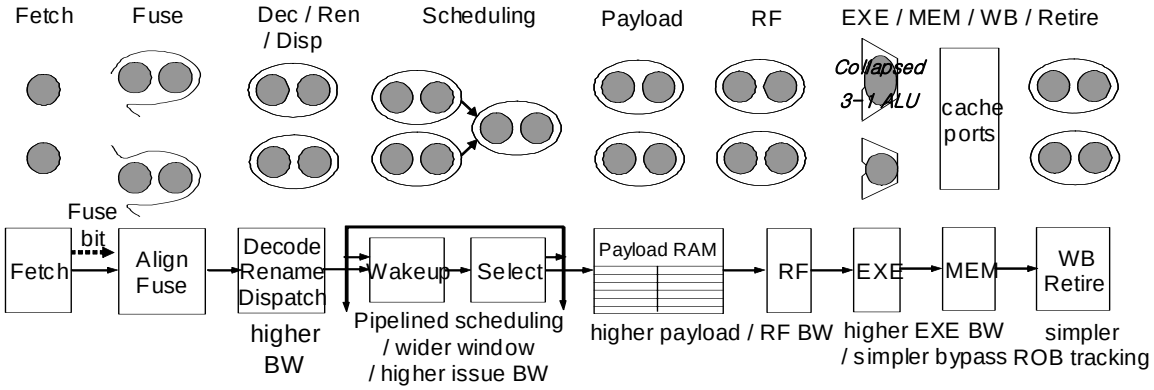
2.4.3 Macro-Op Execution Microarchitecture

The co-designed microarchitecture for the baseline VM has the same basic pipeline stages as a conventional x86 out-of-order superscalar processor (Figure 2.5a). Consequently it inherits most of the proven benefits of dynamic superscalar designs. The key difference is that the proposed macro-op execution microarchitecture can process instructions at the coarser granularity of fused macro-ops throughout the *entire* pipeline.

Several unique features make the co-designed microarchitecture an enhanced superscalar by targeting several critical pipeline stages for superscalar performance. For example, instruction issue logic at the instruction scheduler stage(s), result forwarding network at the execution stage, and code delivery at the instruction fetch stage. The issue stage is especially difficult as it considers the dependences between instructions and needs to wakeup and select instructions. Both wakeup and select are complex operations and they need to be performed in a single cycle [102, 118] to issue back-to-back dependent instructions.

For the DBT optimized macro-op code, fused dependent micro-op pairs are placed in adjacent memory locations in the code cache and are identified via the special "fusible" bit. After they are fetched, the two fusible micro-ops are immediately aligned together and fused. From then on, macro-ops are processed throughout the pipeline as single units (Figure 2.5b).

(a) The macro-op execution pipeline



(b) The macro-op execution overview

Figure 2.5 The macro-op execution microarchitecture

Macro-op fusing algorithm fuses dependent micro-ops at a comparable CISC granularity of the original x86 instructions. However, fused macro-ops are more streamlined and look like RISC 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 better utilized and/or reduced. 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 window slots and issue width can be either effectively increased for better ILP extraction or can be physically reduced for simplicity 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 in conventional designs is especially complex for issuing single-

cycle back-to-back instructions. In our proposed x86 processor VM design, instruction issue can be pipelined in two stages, simply and without performance loss. Another critical stage is the ALU. In our design, two dependent ALU micro-ops in a macro-op can be executed in a single cycle by using a combination of a collapsed three-input ALU [92, 106, 71] and a conventional two-input ALU. Then, there is no need for an expensive ALU-to-ALU operand forwarding network. Rather, there is an entire second cycle where results can be written back to the registers before they are needed by dependent operations.

We improve the pipeline complexity-efficiency in a few ways. (1) Fused macro-ops reduce the number of individual entities (instructions) that must be handled in each pipeline stage. (2) The instruction fusing algorithm strives for macro-op pairs where the first (head) instruction of the pair is a single-cycle operation. This dramatically reduces the criticality of single-cycle issue and ALU operations.

There are other ways to simplify CISC microarchitecture in a co-designed VM implementation. 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 promises a shorter time-to-market. While it is true that the translation software must be validated for correctness, this translation software does not require 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.

2.5 Related Work on x86 Simulation and Emulation

There are many x86 simulation and emulation systems. However, most of these systems are either proprietary infrastructure that are not available for public access, or are released only in binary format that can only support limited exploration for the x86 design space. For example, the Virtu-Tech SIMICS [91] supports full system x86 emulation and some limited timing simulation; the AMD SimNow! [112] is another full system x86 simulator running under GNU/Linux. The simulated system runs both 64-bit and 32-bit x86 OS and applications.

Fortunately, BOCHS [84] is an open-source x86 emulation project. The x86 instruction decode and instruction semantic routines from BOCHS 2.2 are extracted and fully customized to decode and crack x86 instructions into our own designed implementation instruction set. The customized source code is integrated into our *x86vm* framework (Figure 2.1) for full flexibility and support for the desired simulations and experiments.

Dynamic optimization for x86 is an active research topic. For example, an early version of rePLay [104] and recently the Intel PARROT [2] explored using hardware to detect and optimize hot x86 code sequences. The technical details of these related projects will be discussed in Chapter 5 when they are compared with our hardware assists for DBT. Merten *et al* [98] proposed a framework for detecting hot x86 code with BBB (Branch Behavior Buffer) and invoking software handlers to optimize x86 code. However, because the generated code is still in the x86 ISA, the internal sub-optimal code issue is not addressed.

The *x86vm* features a two-stage binary translation system, simple basic block translation for all code when it is first executed and superblock translation for hot superblock optimization. The Intel IA-32 EL [15] employs a similar translation framework. However, there are many variations to this scheme. For example, IBM DAISY/BOA [3, 41, 42] and Transmeta Code Morphing Software (CMS)

[36] use an interpreter before invoking binary translators for more frequently executed code. The Transmeta Efficeon CMS features a four-stage translation framework [83] that uses interpretation to filter out code executed less than (about) 50 times. More frequent code invokes advanced translators based on its execution frequency.

The primary translation unit adopted in *x86vm* is the superblock [65]. A superblock is a sequence of basic blocks along certain execution path. It is amenable for translation dataflow analysis because of its single-entry and multi-exit property. Superblocks are adopted for translation units in many systems, such as Dynamo(RIO) [22], IA-32 EL [15]. However, some translation/compilation systems for VLIW machines supporting predication use tree regions [3, 41, 42] or other units larger than basic blocks as the translation unit.

Chapter 3

Modeling Dynamic Binary Translation Systems

The essence of the co-designed VM paradigm is synergetic hardware and software that implement an architected ISA. In contrast, conventional processor designs rely solely on hardware resources to provide the interface to conventional software. Therefore, it is critical to explore the dynamics of co-designed hardware and software for a clear understanding of VM runtime behavior. The achieved insight should help to improve the efficiency and complexity effectiveness of VM system designs. However, there are few publications that explicitly address the dynamics of translation-based co-designed VM systems.

In this chapter, we first develop an analytical model for staged translation systems from a memory hierarchy perspective. This model captures the first-order quantitative relationships between the major components in a VM system. Then we use this model to analyze VM runtime behavior and strive for an overall translation strategy that balances the translation assignments to different parts of the VM translation system.

3.1 Model Assumptions and Notation

As discussed in Section 2.2 (evaluation methodology), it is easier to appreciate a new design by comparing it with the current best designs. Therefore, we are especially interested in comparing the following two processor design paradigms.

- A reference superscalar paradigm -- the most successful general-purpose microarchitecture scheme in current processor designs. It dominates all the server, desktop, and laptop market and serves as our baseline. In conventional superscalar processors, limited translation is performed in the pipeline front-end every time an instruction is fetched.
- The co-designed VM paradigm -- A hardware/software co-designed scheme that relies on the software dynamic translator to map instructions from the source architected ISA (x86) into the target implementation ISA. The hardware engine then can better realize microarchitecture innovations.

Note that, among the many dynamic binary translation/optimizations systems and proposals, we model systems that use software translation and code caching. In these systems, runtime software translation overhead is a major concern. The size of code cache in a co-designed VM system is typically configured from 10MB to 100MB out of the 512MB to multi-GB main memory size. For example, the Transmeta CMS [36, 82] allocates 16MB for its laptop or mobile device workloads; the IBM DAISY/BOA VMM [3, 41, 42] allocates 100+MB for server workloads.

There are also dynamic binary translation/optimization proposals that incur negligible performance overhead by investing intensive hardware resources for hotspot optimization. These proposals, for example, instruction path coprocessor [25, 26], rePLay [45, 104], and PARROT [2], are mainly designed for dynamic optimizations on the implementation ISA. They do not address code that is not

in hotspots. Additionally, the generated translations are placed in a small on-chip trace cache or frame cache. Therefore, these proposals are not designed for conducting cost-effective, cross-paradigm translations that we are particularly interested in.

The software binary translation that we consider can be either modeled as a whole, i.e. a black box approach, or modeled as a structured object, i.e. a white box approach that distinguishes the major components inside. The specific modeling approach selected for a circumstance depends on the desired level of abstraction.

In our *x86vm* framework, the DBT system of the VM scheme may simply map one basic block at a time in a straightforward way (BBT) for fast startup, or it may perform optimizations on hot superblocks (SBT) for superior steady-state performance. A simple and straightforward way to model the runtime translation overhead is to unify the binary translation behavior as memory hierarchy miss behavior. For example, an invocation of the DBT translation is considered as a miss in the code cache and the miss event handling involves the VM translator. From such a memory hierarchy perspective, we introduce the following notation to model a staged translation system.

M_{DBT} denotes the total number of static instructions that are translated by the DBT system. For the *x86vm* framework, M_{BBT} is used to represent the number of static instructions touched by a dynamic program execution that hence need to be translated first by *BBT*. And, M_{SBT} represents the number of static instructions that are identified as hotspot and thus are optimized by *SBT*. This notation is very similar for memory misses. Assume I instructions have been executed, then the miss rate is $m_{lev} = M_{lev}/I$ where the level, *lev*, can be BBT, SBT or DBT, just as caches, main memory or disk in the memory hierarchy.

Δ_{DBT} stands for the average translation overhead per (translated) architected ISA instruction. In particular, symbols Δ_{BBT} and Δ_{SBT} represent per x86 instruction translation overhead for BBT and

SBT, respectively. The translation overhead can be either measured in terms of cycles (time) or in terms of the number of implementation ISA instructions in which the translators are coded. In this thesis, we use the measure in terms of implementation ISA instructions. Sometimes for comparison purposes, this number is converted to the equivalent number of architected ISA instructions.

3.2 Performance Dynamics of Translation-Based VM Systems

In a conventional system, when a program is to execute, its binary is first loaded from disk into main memory. Then, the program starts execution. As it executes, instructions move up and down the memory hierarchy, based on usage. Instructions are eventually distributed among the levels of cache, main memory, and disk.

In the co-designed VM approach based on software translation and code caching, the program binary (containing the architected ISA instructions) is also first loaded from disk into main memory, just as in a conventional design. However, the architected ISA instructions must be translated to the implementation ISA instructions before they can be executed. The translated code is held in the code cache for reuse until it is evicted to make room for other blocks of translated code. Any evicted translation must then be re-translated and re-optimized if it becomes active again. As a program executes, the translated implementation ISA instructions distribute themselves in the cache hierarchy in the same way as architected ISA instructions in a conventional system.

To simplify the analysis for co-designed VM systems, especially regarding the effects of translation, we identify four primary scenarios.

1. *Disk startup*. This scenario occurs for initial program startup or reloading modules / tasks that were swapped out – the binary needs to be loaded from disk for execution. After memory is loaded, execution proceeds according to scenario 2 below. That is, scenario 1 is the same as scenario 2 below, but with a disk load added at the beginning.

2. *Memory startup*. This scenario models major context switches (or program phase changes) – If a context switch is of long duration or there is a major program phase change to code that has never been executed (or has not been executed for a very long time), then the required translated code may not exist in the code cache. However, the architected ISA code is in main memory, and will need to be (re)translated before it can be executed. This translation time is an additional VM startup overhead which has a negative effect on performance.

3. *Code cache startup / transient*. This scenario models the situation that occurs after a short context switch or short duration program phase change. The translated implementation ISA code is still available in the main memory code cache, but not in the other levels of the cache hierarchy. To resume execution after the context switch (or return to the previous program phase), there are cache misses as instructions are fetched from main memory again. However, there are no instruction translations.

4. *Steady state*. This scenario models the situation where all the instructions in the current working set have been translated and placed properly in the cache hierarchy. The processor is running at "full" speed.

Clearly, scenario 4 *steady state* is the desired case for co-designed VM systems using DBT. Performance is determined mainly by the processor architecture, and the co-designed VM fully achieves its intended benefits due to architecture innovation.

In scenario 3 *code cache transient*, performance is similar in both the conventional processor and VM designs as both schemes fetch instructions through the cache hierarchy, and no translation is required in a co-designed VM. Performance differences are mainly caused by second-order cache effects. For example, the translated code will likely have a larger footprint in main memory, however, the code restructuring for superblock translation will lead to better temporal locality and more efficient instruction fetching.

In contrast, scenario 2 *memory startup* is a bad case and the one where VM startup overhead is most exposed. The translation from architected ISA code (in memory) into implementation ISA code (in the code cache) is required and causes the biggest negative performance impact for dynamic binary translation when compared with a conventional superscalar design.

As noted earlier, scenario 1 *disk startup* is similar to scenario 2, with the added disk access delay. The performance effects of loading from disk are the same in both the conventional and VM systems. Moreover, the disk load time, lasting many milliseconds, will be the dominant part of this scenario. The additional startup time caused by translation will be less apparent and the relative slowdown will be much less in scenario 1 than in scenario 2.

Based on the above reasoning, it is clear that performance analysis of VM system dynamics should focus on scenarios 2 and 4, i.e. *steady-state* performance and *memory startup* performance where VM-specific translation benefit and overhead are prominent.

The steady state performance is mainly determined by the effectiveness of the DBT translation algorithms and the collaboration between co-designed hardware processor and translated/native software code. Chapter 4 addresses the translation algorithms and Chapter 6 emphasizes the collaboration and integration aspects for VM systems.

For VM-specific performance modeling and translation strategy research, translation-incurred transient behavior during VM startup phase is the key and will be focused on in this chapter. Particularly, we emphasize the memory startup scenario, that is, when we analyze startup performance, we start with a program binary already loaded from disk, but with the caches empty, and then track startup performance as translation and optimization are performed concurrently with execution.

A set of simulations are conducted for the *memory startup* scenario to compare performance between the reference superscalar processor and two co-designed VM systems that rely on software for DBT. The first staged translation system uses BBT followed by SBT, and the second uses interpretation followed by SBT. All systems are modeled with the *x86vm* framework and the specific configurations are the same as in Table 5.4 later in Section 5.5. The simulations start with empty caches and run 500-million x86-instruction traces to track performance. The traces are collected from the ten Windows applications in the Winstone2004 Business suite. The total simulation cycles range from 333-million to 923-million cycles for the reference superscalar.

The simulation results are averaged for the traces and graphed in Figure 3.1. IPC performance is normalized with respect to the steady state reference superscalar IPC performance. The horizontal line across the top of the graph shows the VM steady state IPC performance gain (8% for the Windows benchmarks).

The x-axis shows execution time in cycles on logarithmic scale. The y-axis shows the harmonic mean of their *aggregate* IPC, i.e. the total instructions executed up to that point divided by the total time. As mentioned, at a given point in time, the aggregate IPC reflect the total number of instructions executed (on a linear scale), making it easy to visualize the relative overall performance up to that point in time.

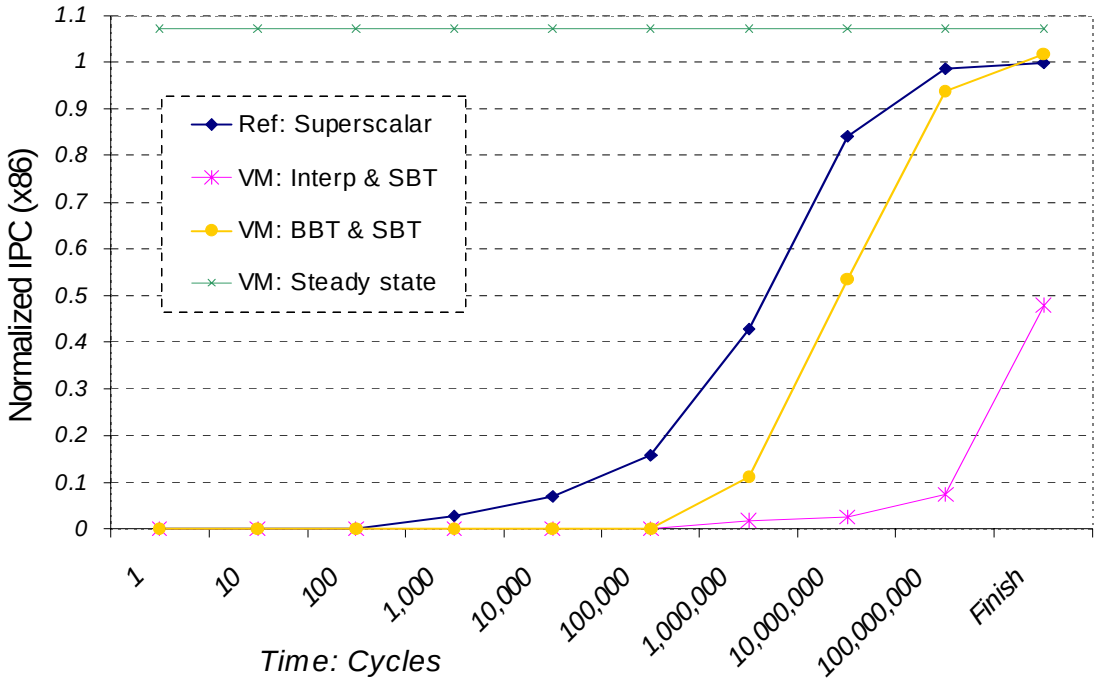


Figure 3.1 VM startup performance compared with a conventional x86 processor

The breakeven point (as used in Chapter 1 for startup overhead) is the time it takes a co-designed VM to execute the same number of instructions as the reference superscalar processor. This is opposed to the point where the instantaneous IPCs are equal, which happens much earlier. The crossover, or breakeven, point occurs later than 100-million cycles for the baseline VM system using staged BBT followed by SBT. And this co-designed VM system barely reaches half the steady state performance gains (4%) before the traces finish.

For the co-designed VM using interpretation followed by SBT, the startup performance is much worse. The hotspot threshold for switching from interpretation to SBT is 25 (as derived using the method described below in Section 3.3). After finishing the 500-million instruction traces, the aggregate performance is only half that of a conventional superscalar processor.

Clearly, software DBT runtime translation overhead affects VM startup performance when compared with a conventional superscalar processor design, especially for a startup periods less than 100-million cycles (or 50 milliseconds in a 2 GHz processor core). At the one-million-cycle point, the baseline VM system has executed only one fourth the instructions of the reference conventional superscalar implementation.

3.3 Performance Modeling and Strategy for Staged Translation

In a two-stage translation system consisting of BBT and SBT, the emulation starts with simple basic block translation. Dynamic profiling is used to detect hot code regions. Once a region of code is found to be "hot", it is re-organized into superblock(s) and is optimized. Therefore, translation overhead is a function of two major items. (1) The number of static instructions touched by a dynamic program execution that therefore need to be translated first by BBT, i.e. M_{BBT}. (2) The number of static instructions that are identified as hotspot and thus are optimized by SBT, i.e. M_{SBT}. The symbols Δ_{BBT} and Δ_{SBT} are used to represent per x86 instruction translation overhead for BBT and SBT, respectively. Then, for such a system, the VM translation overhead is:

$$\textit{Translation overhead} = M_{BBT} * \Delta_{BBT} + M_{SBT} * \Delta_{SBT} \qquad \text{(Eq.1)}$$

Clearly, M_{BBT} is a basic characteristic of the program's execution and cannot be changed. Thus, a feasible way to reduce BBT overhead is to reduce Δ_{BBT}, and for this goal, we will propose hardware assists in Chapter 5. Regarding the SBT component, we argue that good hotspot optimizations are complex and need the flexibility advantages of software. A hardware implemented optimizer (at least for the optimizations we consider) would be both complex and expensive. Chapter 4 will strive for both efficient and effective translation algorithms. Fortunately, for most applications, the hotspot code is a small fraction of the total static instructions. Furthermore, the hotspot size, M_{SBT}, is sensitive

to the hot threshold setting. Therefore, we need to explore a balanced trade-off regarding the hot threshold setting, which not only reduces SBT overhead by detecting true hotspots, but also collects optimized hotspot performance benefits via good hotspot code coverage.

Our evaluation of this trade-off uses a specialized version of the model proposed for the Jikes RVM java virtual machine [11]. Let p be the speedup an optimized superblock can achieve over the simple basic block code. Also let N be the number of times a given instruction executes and let t_b be the per instruction execution time for code generated by BBT. Then, to break even, the following equation holds. (This assumes that the optimizer is written in optimized code and its overhead Δ_{SBT} is measured in terms of architected ISA instructions.)

$$N * t_b = (N + \Delta_{SBT}) * (t_b / p) \Rightarrow N = \Delta_{SBT} / (p - 1) \quad \text{(Eq.2)}$$

That is, the breakeven point occurs when the number of times an instruction executes is equal to the translation overhead divided by the performance improvement. In practice, at a given point in time, we do not know how many times an instruction will execute in the future, however. So, this equation cannot be applied in an *a priori* fashion. In the Jikes system, it is assumed that if an instruction has already been executed N times, then it will be executed at least N times more. Hence, the value N as given in Equation 2 is used as the threshold value.

In our VM scheme, we set the hot threshold that triggers SBT translation based on Equation 2 and benchmark characteristics. To calculate the hot threshold based on the equation, we first determine the values of the equation parameters. For our VM system, we have measured Δ_{SBT} to be 1152 x86 instructions (approximately 1200) and p is 1.15 ~ 1.2 for the WinStone2004 traces. That is, SBT optimized code runs 15 to 20 percent faster than the code generated by BBT. Then to break even, Equation 2 suggests that N should be at least 1200/0.15 = 8000 for WinStone2004-like workloads.

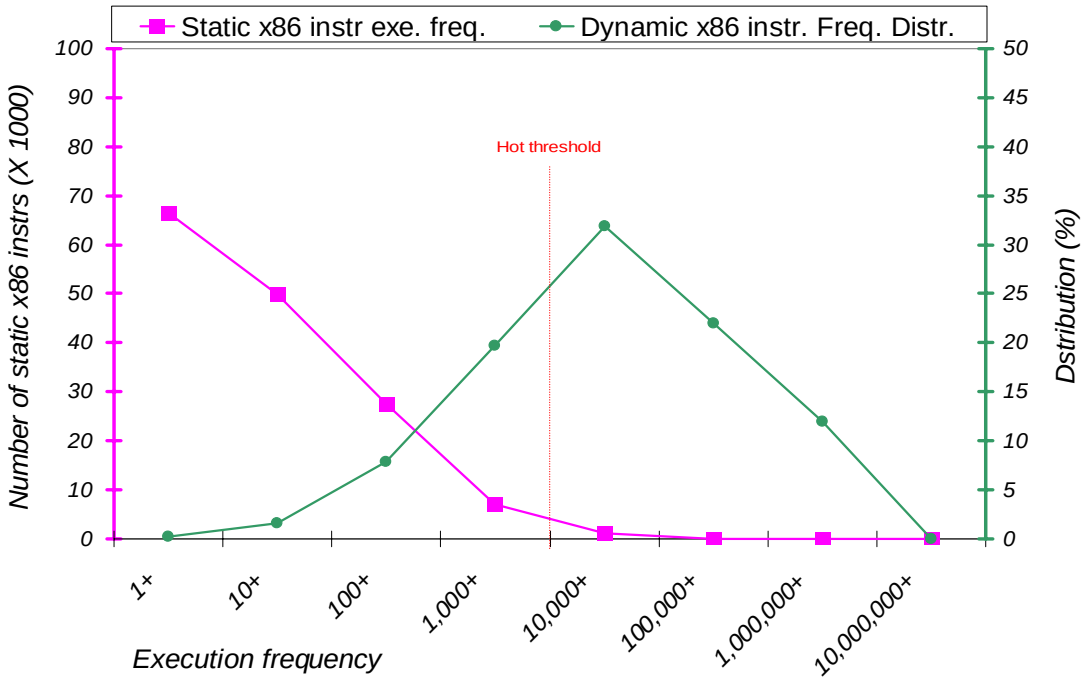


Figure 3.2 Winstone2004 instruction execution frequency profile

(100M x86 instruction traces)

To illustrate the reasoning and the motivation for a relatively high hotspot threshold, we have conducted benchmark characterization for the Windows applications. We used data averaged over the traces of length 100-million x86 instructions collected from the ten Winstone2004 Business suite applications. The x-axis of Figure 3.2 is instruction counts (frequency). The left y-axis shows the number of static x86 instructions that are executed for the number of times marked on the x-axis. The threshold execution count (8000) is marked with a red vertical line. By using the left y-axis, we see that only 3K (determined by adding the data points of the static instruction curve that are to the right of the hot threshold line) of the static instructions have exceeded the hotspot threshold at the time 100 million instructions have been executed. It is clear from the figure that, for these benchmarks, only a small fraction of executed static instructions are classified as being in hotspots.

The right y-axis shows the distribution function of total dynamic x86 instructions. For example, the peak point of the distribution curve shows that 30+% of all dynamic instructions execute more than 10K times, but less than 100K times. This curve rises and then falls off because the total dynamic instruction count for the simulations is 100 million. If the programs run longer than 100 million instructions, then this curve would continue to rise and the peak would shift to the right toward higher execution frequency. It is clear from the figure that for a hot threshold on the order of thousands, the hotspot coverage (the percentage of instructions executed from the optimized hotspot code) is fairly modest for these short startup traces. However, the hotspot code coverage will be significantly improved once benchmarks are run much longer as in most realistic cases. For example, the hotspot coverage in these short traces is about 63%. The coverage will increase to 75+% for 500M instruction runs. And real applications run tri/billions of instructions.

Returning now to Equation 1, the average value of M_{BBT} is 150K static x86 instructions (this can be determined by adding the data points of the static instruction curve) and the average value of M_{SBT} is 3K (adding the data points of the static instruction curve that are to the right of the hot threshold line). Assuming Δ_{BBT} = 105 native instructions (as measured in our baseline VM system) and Δ_{SBT} = 1674 native instructions (equivalent to the 1152 x86 instructions above), then we infer that the BBT component is 105 * 150K = 15.75M native instructions, or equivalent to 10.86M x86 instructions, and the SBT component is 1674 * 3K = 5.02M native instructions, or equivalent to 3.46M x86 instructions. In other words, based on the fact that 100M workload x86 instructions are executed, the analytical model projects that the BBT overhead is roughly 10% and the SBT overhead is about 3%. Therefore, in our VM system, BBT causes the major translation overhead, and this is the overhead Chapter 5 will tackle with hardware accelerators. Also because BBT translation is a simpler operation, it offers more opportunities for hardware assists.

It is important to note that an advantage of the co-designed VM paradigm is its synergetic hardware/software collaboration. This unique capability shifts the trade-off for the translation strategy. And it is not always feasible to achieve this capability in VM systems that aim at porting user-mode software across different platforms. For example, the Intel IA-32 EL instruments extra code in BBT translations to collect program execution profile and apply certain translation rules to guarantee precise state can be easily recovered should an exception occur. This profiling code and rules cause significant extra overhead and slow down the BBT translations for emulation speed. In contract, a co-designed VM system employs cost-effective hardware assists to detect program hotspot and to accelerate the critical part of DBT translation. BBT translations in co-designed VM can be more efficient and the hotspot optimization overhead can thus be reduced by translating only confi-dent hotspots. Once hardware assists for BBT are deployed, BBT translation becomes much cheaper and flushing the code cache for BBT becomes inexpensive. Then, using interpreters to filter out code that executes very infrequently (less than 20 times) becomes less attractive (Perhaps, Transmeta uses the first stage of interpretation to filter out the leftmost data point(s) in Figure 3.2, which stands for instructions that execute less than 10 times but take up a big section of the code cache).

In a co-designed VM system, the overall translation strategy strives for (1) balanced translation assignments among the major components and (2) an optimal division between the co-designed hardware and software. In our *x86vm* framework, the translation strategy for the two-stage translation system is based on the analytical model (Equations 1 and 2). Specifically, our translation strategy determines the hotspot threshold based on Equation 2 and application characteristics. This strategy also suggests that hardware primitives should significantly accelerate BBT translation, which is on the critical execution path.

3.4 Evaluation of the Translation Modeling and Strategy

The analytical model that estimates DBT translation overhead (Equation 1) is evaluated by comparing its projections with performance simulations. The simulations are conducted for the baseline VM system running the Windows application traces. Figure 3.3 shows the runtime overhead for both the BBT and the SBT relative to the entire execution time. Clearly, BBT translation overhead is more critical and is measured at 9.6%, close to the 10% prediction based on the analytical model. The SBT hotspot optimization overhead is also fairly significant and is measured at 3.4%, close to the 3% projected by the equation.

Furthermore, the BBT translation must be performed on the critical execution path because in our VM system there is no other way to emulate a piece of code that is executed for the first time. On the other hand, the SBT optimization is relatively more flexible because the BBT translation is still available. This flexibility leads to more opportunities to hide SBT optimization overhead.

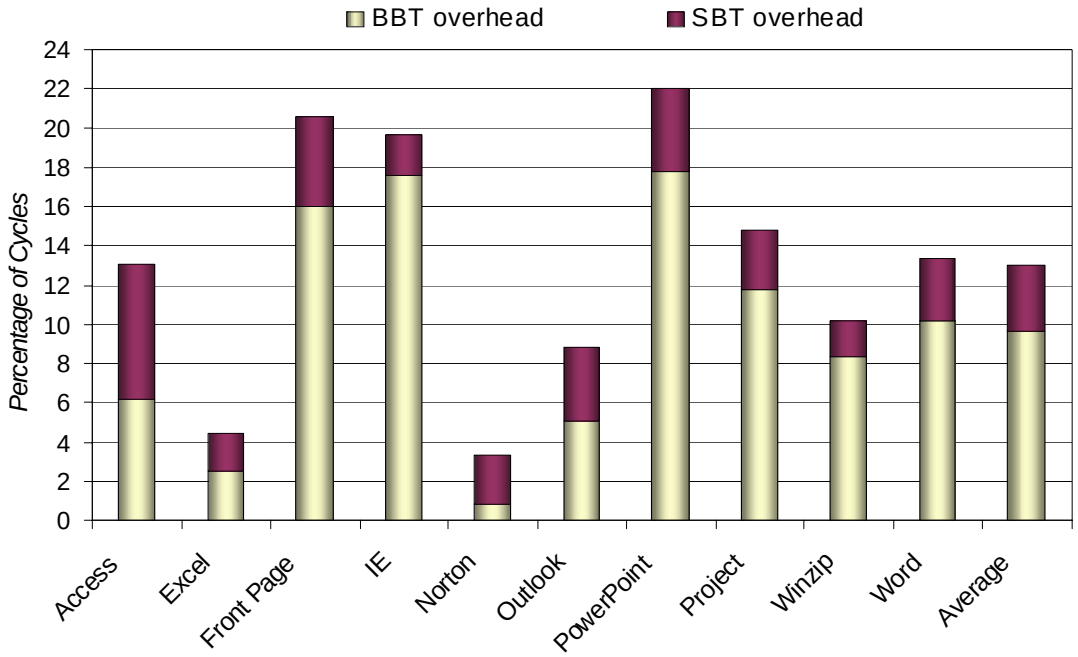


Figure 3.3 BBT and SBT overhead via simulation

Figure 3.3 corroborates the observation derived from Figure 3.1 that runtime overhead affects the VM startup performance significantly; it can not be simply assumed to be negligible at least for the applications that we enumerated in Chapter 1.

The analytical model that estimates the appropriate thresholds (Equation 2) for triggering more advanced translation stages is evaluated by observing how the VM system performance varies as the hot threshold changes around the ranges predicted by the equation. Thresholds too low will cause excessive translation overhead while thresholds too high reduce the hotspot code coverage and thus the optimization benefits. Typically, as the hot threshold increases, the hotspot size decreases more quickly than the hotspot coverage.

Figure 3.4 plots the VM IPC performance versus hot threshold trend for each benchmark traces from the WinStone2004. As the hot threshold increases from 2K to the optimal point (8K, determined

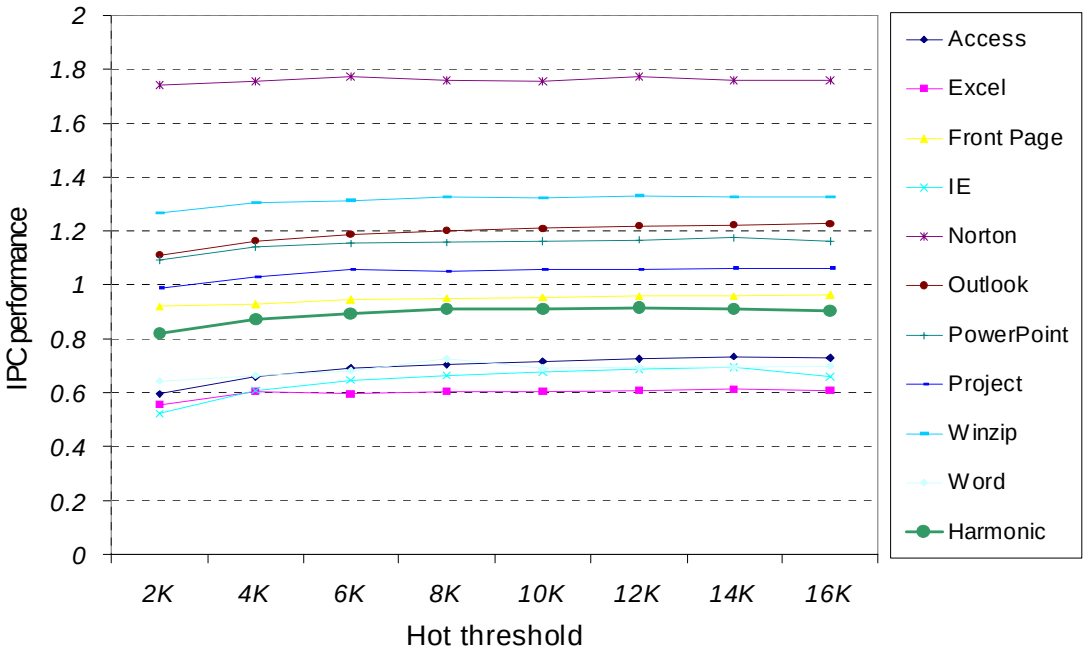


Figure 3.4 VM performance trend versus hot threshold settings

by Equation 2), the VM performance clearly increases. After the optimal point, the VM performance gradually decreases.

Figure 3.4 verifies that the analytical model calculates a fairly balanced hot threshold based on averages for the whole benchmark suite. However, the best performance point for each specific benchmark is different. Perhaps, this suggests an adaptive threshold for each application or even each program phase for optimal performance and efficiency.

An assumption in the previous discussion is that the different memory hierarchy behavior (VM versus baseline superscalar) causes only second-order performance effects. This assumption is the basis for the simple VM system analytical model, which only models translation behavior. Although this assumption matches common sense, supportive data and analysis will now be given to validate this assumption.

Compared with a conventional superscalar processor, the co-designed VM system has three major performance implications.

1. The VM has a better steady state performance (clock speed and/or IPC, CPI. Only the IPC advantage is shown in Figure 3.1). The better performance comes from the implementation ISA and microarchitecture innovations.
2. The VM pays the extra translation overhead, which is modeled as: $M_{DBT}(i)*\Delta_{DBT}$.
3. The VM has a different memory hierarchy behavior, mainly for instructions. The memory system behavior can be modeled as $M_{L2}(i)*\Delta_{L2} + M_{mem}(i)*\Delta_{mem}$.

Similar to the notation for translation modeling, M_{L2} and M_{mem} are the number of *instruction fetch* misses to L2 cache and main memory respectively. Δ_{L2} and Δ_{mem} represent the miss penalty for a miss to L2 cache and main memory accordingly. For a specific memory hierarchy level, the

Table 3.1 Benchmark Characterization: miss events per million x86 instructions

WINSTONE2004	VM: M_{BBT}	VM: M_{HST}	VM: M_{L2}	VM: $M_{C\$}$	REFERENCE SS : M_{L2}	REFERENCE SS : M_{MEM}
Access	923	68.4	9802	109	12389	146
Excel	605	38.6	15760	1191	15402	1368
Front Page	2791	51.4	5668	413	3956	338
Internet Explorer	3872	29.8	25612	987	16191	688
Norton Anti-virus	57	12.4	8.1	4.2	20.8	3.8
Outlook	222	25.6	178	16.6	174	13.9
Power Point	2469	38.3	4727	463	3758	378
Project	1968	32.4	7504	204	5178	157
Win-zip	824	11.6	2930	171	2249	160
Word	1299	26.7	1902	98	1608	86
Average	**1503.0**	**33.5**	**7409.0**	**365.7**	**6092.5**	**333.9**

instruction miss penalty tends to be nearly constant [73] and is closely related to the access latency for the memory level that services the miss. Typically, the access latency is easily determined via simulations.

We collect data to compare the last two implications and determine if memory behavior difference causes only second-order effects. Table 1 shows miss rates for the listed benchmarks running 100M x86-instruction traces. The simulations are configured similar to Figure 3.1 and 3.2.

Two factors affect the VM system memory hierarchy behavior, (1) code straightening via hot superblock formation for hotspot optimization, (2) code expansion and translator footprint. Because we translate x86 instructions into a RISC-style ISA, there is some code expansion. In general, the 16/32b fusible ISA binary has a 30~40% code expansion over the 32-bit x86 binaries. Table 3.1 shows that the VM system causes more instruction cache misses to the L2 cache, 0.0013 more misses per x86 instruction. Assuming a typical L2 cache latency of 10-cycle, the extra instruction cache misses lead to a 0.013 CPI adder for the VM system. In real processors, some of the L1 misses can be tolerated. The latency to the DRAM main memory is much longer, well over 100-cycles. The VM system causes 0.000032 more misses to memory per x86 instruction. Therefore, assuming a 200-cycle

memory latency, the extra misses to main memory will cause another 0.0064 CPI adder to the VM system CPI. In summary, the different memory hierarchy behavior will cause an overall 0.02 CPI adder to the VM system.

Now consider the translation overhead, there are two sources of translation overhead, the BBT overhead and the SBT overhead, $M_{BBT}(i) \cdot \Delta_{BBT} + M_{SBT}(i) \cdot \Delta_{SBT}$. On average, about 1.5 out of every 1000 x86 instructions executed need to be translated by BBT in the tested traces. Assuming a typical BBT translation overhead of 100-cycles per x86 instruction, the CPI adder due to BBT translation is 0.15. Table 3.1 shows about 33.52 instructions are identified as hotspot per million dynamic x86 instructions. Assuming a 1500 cycle SBT overhead per x86 instruction in the VM system, the data suggests that the SBT overhead leads to another CPI adder of about 0.05 for the VM model. To summarize, the VM translation overhead causes an overall 0.2 CPI adder to the VM system for the program startup phase.

It is clear from the above data and analysis that the DBT translation overhead is an order of magnitude higher than the extra memory miss-penalty. This analysis backs up the assumption that the different memory hierarchy behavior causes only second-order performance impact.

3.5 Related Work on DBT Modeling and Strategy

A first-order performance model for dynamic superscalar processors is proposed by Karkhanis ans Smith in [73]. It determined the linear relationship for the overall processor performance among major performance factors such as branch prediction, L1 cache misses. Thus, the CPI model of a superscalar processor simply adds the first-order performance factors to the base ideal model. Other related work on analytical modeling of processor pipelines is also surveyed in [73]. Our VM performance modeling extends their models to include dynamic translation factors and this becomes feasible by observing the translation system behavior from a memory hierarchy perspective.

Transmeta published limited information about their co-designed x86 processors [82, 83, 122], which are probably the only co-designed VM products currently being delivered. Performance modeling of the Transmeta CMS system is not disclosed. The translation strategy [82, 83] is described as a four-stage emulation system that starts with an interpreter. The trade-offs regarding setting the hot thresholds (for triggering the higher stages for more advanced optimizations), however, are not disclosed. The CMS translation strategy also adopted a hardware assist scheme [83] that accelerates the first stage, the interpreter in the Efficeon CMS system. There are no official benchmark results for the Transmeta x86 processors.

IBM DAISY/BOA project(s) published performance data [3, 41] for SPEC CPU and TPC-C benchmarks. Since SPEC CPU benchmarks have very good code reuse behavior, especially for full reference runs, the translation overhead is negligible. TPC-C is one of the few benchmarks in their results that demonstrate significant translation overhead. Performance modeling and consequently reducing translation overhead was not emphasized in the DAISY/BOA research. DAISY also used interpretation for initial emulation and the hot threshold for invoking the binary translator is set on the order of tens due to the slow emulation speed of an interpreter.

The ILDP co-designed VM research [76~79] takes a similar (to DAISY) translation strategy and simply assumes translation overhead is negligible for most applications.

The situation is similar for other VMs similar to the co-designed VM domain. For example, developers of commercial products (such as the Intel IA-32 EL [15]) have not published their translation trade-off, modeling, and strategies. However, the IBM Jikes RVM java virtual machine research [11] proposed a similar method to set the thresholds that trigger more advanced optimizations for hot java methods. Their equation $T_j = T_i * S_i / S_j$ is essentially the same as our Equation 2, except that it is expressed for dynamic java compilation setting.

Chapter 4

Efficient Dynamic Binary Translation Software

The most distinctive feature of the co-designed virtual machine paradigm is its ability to take advantage of concealed, complexity-effective software. The co-designed VM software is primarily for runtime translation from the architected ISA to the implementation ISA, and can potentially enable other attractive features. Software ISA mapping allows intelligent translation and optimization at the cost of runtime overhead. Therefore, it is important for the VM software not only to generate efficient native code for high performance, but also to run very efficiently itself (to reduce translation overhead).

In this chapter, I discuss specifically how x86 instructions are translated (and optimized) by software DBT to the fusible ISA code. First, the translation procedure is described in Section 4.1. Then, I elaborate on the translation unit formation in Section 4.2 and machine state mapping from the x86 to the fusible ISA architected state in Section 4.3. The key translation algorithms that discover appropriate dependent instruction pairs for fusing and schedule these dependent pairs together are detailed in Section 4.4 and 4.5. Section 4.6 discusses how to perform simple and straightforward BBT translation targeting fast VM startup. Finally, Section 4.7 evaluates software translators and Section 4.8 discusses related work on software dynamic binary translation.

4.1 Translation Procedure

Dynamic translation algorithms are highly dependent on the particular combination of the architected ISA and the implementation ISA. In the *x86vm* framework, with a fusible ISA and *macro-op execution* pipeline, the binary translator includes the following steps.

1. *Translation unit formation*. The translation unit selected in the framework is the superblock [65]. Program hotspots are detected by simple hardware profiler, for example as that proposed by Merten *et al*. [98], and are formed into superblocks.

2. *IR Generation.* The x86 instructions in a superblock are decoded and cracked into a RISC style intermediate representation (IR). Memory access (load / store) instructions and other instructions with embedded long (32-bit) immediate values are transformed into an IR form that preserves the long immediate values. The purpose is to keep the original semantics of the x86 instructions without concern for encoding artifacts.

3. *Machine state mapping*. All frequently-used x86 register names are mapped to the fusible ISA registers in a straightforward way (Section 4.3). This mapping is employed in the previous step in fact. Additionally, the translator scans superblocks for long immediate values and counts frequencies in a small temporary table. Then, it performs a value clustering/locality analysis to allocate frequent immediate values to registers as described in Section 4.3.

4. *Dependent Graph Construction*. Register value dependence is analyzed; this includes x86 condition code dependences. A simple dynamic dependence graph is then constructed for the superblock and is maintained in place with its IR data structure.

5. *Macro-op Fusing*. Appropriate dependent instructions are discovered and paired together to form fused macro-ops (Section 4.4 and 4.5). Dependent instruction 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. Dependences due to x86 condition codes are handled as normal data dependences and many fused pairs are in fact formed around condition codes.

6. *Register Allocation*. Before this step all registers are identified with pseudo register numbers. To allow precise state recovery as described by Le [85], actual register allocation needs to be done at this point. As instructions are reordered, register live ranges are extended to allow precise state recovery. Permanent register state mapping (Section 4.3) is maintained at all superblock boundaries.

7. *Code Generation*. The fusible ISA instructions with fusing information are generated and encoded. Then, the translated superblock is stored in the code cache and registered with the VMM for native execution.

The above translation and optimization procedure is typically performed only for program hotspots. Otherwise, a simpler light-weight translator (Section 4.6) is applied in order to avoid excessive runtime translation overhead.

4.2 Superblock Formation

As noted above, a superblock is a sequence of basic blocks along a certain execution path. The typical size of a superblock is several basic blocks. And a nice property of a superblock that simplifies optimization is that it has only one entry point (though there are multiple exit points). This property makes dataflow or dependence analysis within the superblock simpler than otherwise. A disadvantage of superblocks, however, is that they often result in replicated tail code – a constituent

basic block of a superblock that is not the entry point can also be part of another superblock, thus causing code replication.

For superblock formation, three issues are especially important, the superblock start point, the execution path to follow, and the superblock termination conditions. In the *x86vm* framework, the superblock start point is determined via a hardware profiler that determines when the execution frequency of a basic block (or equivalently a branch target) has exceeded a pre-determined *hot threshold.* Once such a hot threshold has been reached, the most frequent execution path is followed via a profiler counter table. When a superblock is ended depends on the termination conditions defined in a specific system. In the *x86vm*, the termination conditions are, (1) indirect jumps such as function returns, indirect calls or braches; (2) backward conditional branches; (3) a cycle along the path is detected; (4) the execution path already exceeds a defined maximum superblock size, which is 512 fusible ISA instructions cracked from the x86 instructions.

4.3 State Mapping and Register Allocation for Immediate Values

To emulate an architected ISA efficiently, it is important to map the registers in the architected ISA to the native registers in the implementation ISA. Otherwise, extra loads and stores are needed. Any additional memory operations are especially detrimental to performance.

To generate efficient native code for emulating the x86 ISA, the binary translator employs a permanent register state mapping. The sixteen x86 general-purpose registers (the translator is designed with the 64-bit x86 support in mind. However, it is not benchmarked with 64-bit binaries for this thesis) are mapped to the first sixteen of the 32 general purpose integer registers, R0 through R15. This standard mapping is maintained at superblock boundaries. For the first eight x86 registers, we use an x86-like notation for readability (e.g. Reax corresponds to x86 eax). The other eight registers, R8 to R15, are mapped directly.

R31 is the zero register. Because most x86 binaries have a fairly significant number of zero immediate values, the zero register can reduce dynamic instruction count considerably. Registers R24 to R30 are used mainly by the virtual machine software for binary translation, code cache management, runtime control and precise state recovery, etc. Using a separate set of registers for VM software can avoid the overhead of "context switches" between the VM mode and the translated native mode.

Registers R16 to R23 are temporary/scratch registers. They are mostly used for providing local communication between two operations cracked from the same x86 instruction. The VM mode can also use these scratch registers for temporary values before transfer control to native code. These scratch registers never need to be saved across superblocks or modes.

The x86 SSE (1,2,3) registers are mapped to the first sixteen F registers, F0 through F15, of the 32 native 128-bit F registers. All x87 floating point and MMX registers are mapped to F16 to F23. The rest of the F registers are FP or media temporary/scratch registers.

The x86 condition code and FP/media status registers have direct correspondence with registers in the fusible ISA, which is designed for implementing the x86 efficiently. For example, the x86 PC is maintained in one of the VM registers for VM runtime controls and precise x86 state.

Due to the relatively small number of general-purpose registers in the x86 ISA (especially the 32-bit x86 workloads that are benchmarked in this project), x86 binaries tend to have more immediate values than a typical RISC binary. For example, about 10% of all x86 memory access instructions [16] have embedded memory addresses as either an absolute address or a base address. 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 each long immediate value using at least two extra instructions.

To reduce the code expansion (and instruction count) that would result, the VM binary translator collects these long immediate values within a superblock, analyzes the differences (deltas) among the values to identify values that "cluster" around a central value. If such a case is detected, then the translator converts a long immediate operand to either a register operand (if the value has already been loaded) or 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). For the example aforementioned, an absolute addressing instruction can often be converted to a register indirect or register displacement addressing instruction. Combined with other frequent immediate values converted by this algorithm, the dynamic fusible ISA instruction count can be reduced by several percent.

4.4 Macro-Op Fusing Algorithm

The proposed co-designed VM system features *macro-op execution* which enables more efficient execution of the translated native code by significantly increasing the effective pipeline bandwidth without increasing critical pipeline resource demands. In fact, macro-ops reduce the complexity of critical pipeline stages such as the instruction scheduler, register ports, execution stage and result operand forwarding logic.

Clearly, the key translation/optimization for SBT is a fusing algorithm that first discovers *appropriate* dependent instruction pairs and then fuses them into macro-ops. The objectives for the macro-op fusing algorithms are: (1) to maximize the number of fused dependent instruction pairs, especially those with critical dependencies, and (2) to be simple and fast to reduce translation overheads.

A number of fusing heuristics are targeted at the macro-op execution engine. The first heuristic concerns the *pipelined scheduler*: this heuristic always prioritizes single-cycle ALU instructions

for fusing, especially for the head of a pair. A multi-cycle instruction will not see IPC losses from pipelined scheduling logic, so there is relatively little value in prioritizing it.

The second heuristic addresses the *criticality* of the fused dependences: it first tries to pair up instructions that are close together in the original x86 code sequence. The rationale is that these close dependent instruction pairs are more likely to need 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. For example, they are more amenable for extending register live ranges to provide precise state recovery [85] if there is a trap.

The third heuristic considers pipeline *complexity*. It requires the algorithm to fuse dependent instruction pairs that have a combined total of two or fewer unique input register operands. This ensures that the fused macro-ops can be easily handled by most conventional pipeline stages such as the register renaming stage, instruction scheduling logic, and register file access ports.

To develop the fusing algorithm for fast runtime translation, we concentrate on algorithms that fuse macro-ops via linear scans though the IR instructions either once or multiple times, if necessary. For each linear scan, there are two possibilities, either a forward scan or a backward scan.

After the data dependence graph is constructed, a *forward scan* algorithm considers instructions one-by-one as candidate *tail* instructions. For each potential tail, the algorithm 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.

Alternatively, a *backward scan* algorithm traverses from the penultimate 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.

```
Algorithm: Macro-op Fusing
In :    micro-op sequence
Out:    micro-op sequence marked with fusible bit info)
1. for (int pass = 1, pass <=2; pass++) {
2.      for (each micro-op uop from the 2nd to the last) {
3.           if (uop already fused) continue;
4.           if (pass == 1 and uop multi-cycle, e.g. mem-ops) continue;
5.           look backward via dependency edges for its head candidate;
6.           if (heuristic fusing tests pass) mark as a new fused pair;
7.      }
8. }
```

Figure 4.1 Two-pass fusing algorithm in pseudo code

Neither a forward linear scan nor the backward linear scan algorithm in the dynamic binary translator is necessarily optimal. However, we have found that they are near-optimal. In cases I have manually inspected, they capture well over 90% of the possible fusible pairs. And preliminary algorithms and evaluations indicate that a forward scan performs slightly better than a backward scan.

Note that the direction for searching fusing candidate dependence edges in these algorithms is always opposite to the scan direction, so we call this an *anti-scan (direction) fusing heuristic*. The rationale for this will be explained below.

A *forward two-pass scan algorithm* was eventually developed to discover macro-ops quickly and effectively (Figure 4.1). A two-pass scan is selected because it naturally honors the *pipelined scheduler* and *criticality* heuristics without losing efficiency. The code lines specific to the two-pass fusing algorithm are highlighted in the figure.

After constructing data dependence graph, the first *forward scan* pass only considers single-cycle instructions one-by-one as candidate *tail* instructions. Single-cycle instructions are prioritized because the dependence between ALU operations is often more critical and is easier to determine.

A second scan is performed to consider multi-cycle instructions such as loads and stores as fusing candidate tails. The goal is to fuse as many macro-ops as possible; in this scan criticality is less of an issue.

For a pair of candidate dependent instructions, there is a suite of fusing tests to be performed. Only if all tests are passed, can the pair be marked as fused macro-op. For example, the *complexity* constraint requires that any pair of candidate instructions that have more than two unique source register-operands can not be fused. There are other important constraints.

Code scheduling algorithms that group dependent instruction pairs together need to maintain all the original data dependences. However, some data dependence patterns inhibit such code re-ordering. For example, consider the case where a head candidate for a given tail produces a value for both its tail candidate and another instruction that separates them in the original code sequence (Figure 4.2a). If the instruction in the middle (N) also produces an operand for the tail, then making the tail and head consecutive instructions (as is done for fusing) must break one of the dependences between the candidate pair and the "middle" instruction. Note that in Figure 4.2, 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 4.2b and 4.2c). However, analysis of these cases is not as straightforward as in Figure 4.2a. Therefore, we need a mechanism to avoid breaking data dependences when we reorder instructions for grouping dependent pairs together.

A nice property of the anti-scan fusing heuristic is that it assures that the pairing pattern shown in Figure 4.2b will not occur. In the case shown in Figures 4.2b and 4.2c, 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 Figure 4.2c.

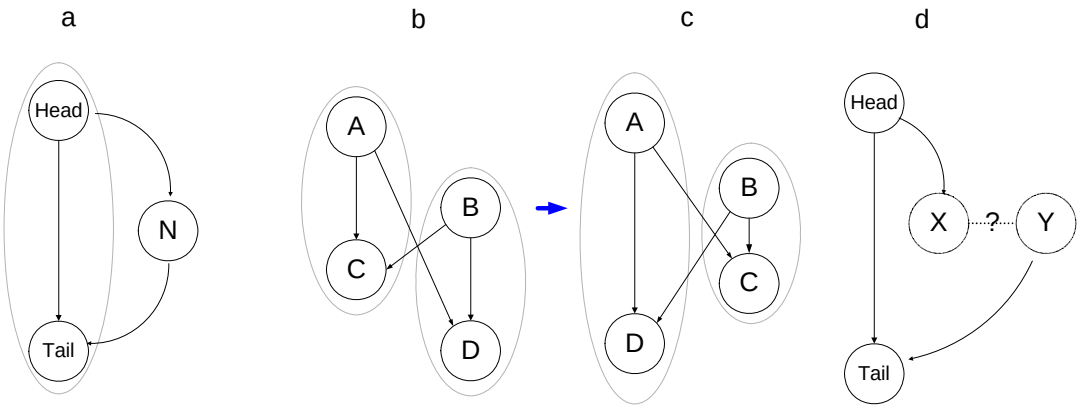


Figure 4.2 Dependence Cycle Detection for Fusing Macro-ops

There is an advantage to reducing cross dependences in the case shown in Figure 4.2c versus the case in Figure 4.2b. That is, while considering candidate instructions for pairing, the anti-scan fusing heuristic enables the algorithm to consider only instructions *between* the candidate head and tail for detecting any potential dependence cycles. This reason is that the anti-scan fusing heuristic guarantees that B and C are first paired before A and D can be considered. In contrast, if the case shown in Figure 4.2b can occur, then the algorithm has to analyze all the dependent instructions either before the head or after the tail.

Figure 4.2d models the general case for detecting dependence cycles, of which Figure 4.2a and 4.2c are special cases. Under the anti-scan fusing heuristic, the data dependence cycle detection algorithm only considers nodes between the candidate head and tail when looking for potential cycles (which inhibit fusing). Therefore, the anti-scan fusing heuristic provides an efficient mechanism to avoid dependent cycles during fusing. Consequently, it ensures the linear complexity of the fusing algorithm. Otherwise, the complexity would be quadratic for an algorithm that considers all the dependent instructions beyond the head and tail candidates.

(a) x86 assembly

```
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]
```

(b) micro-operations

```
1. ADD   Reax, Redi, 1
2. ST    Reax, mem[R18]
3. LD.zx Rebx, mem[Rebp + Recx << 1]
4. AND   Reax, 0000007f
5. ADD   R21,  Reax, Resi
6. LD    Redx, mem[R21 + 0x7c]
```

(c) Fused macro-ops

```
1. ADD   R20,  Redi, 1    ::    AND    Reax, R20, 007f
2. ST    R20,  mem[R18]
3. LD.zx Rebx, mem[Rebp + Recx << 1]
4. ADD   R21,  Reax, Resi ::    LD     Rebx, mem[R21 + 0x7c]
```

Figure 4.3 An example to illustrate the two-pass fusing algorithm

Figure 4.3 illustrates the dynamic binary translator fusing dependent pairs into macro-ops. In Figure 4.3a, a hot x86 code snippet is identified from 164.gzip in SPEC2000. Then, the translator cracks the x86 binary into the RISC-style instructions in the fusible implementation ISA, as shown in Figure 4.3b. 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 4.3c). 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; 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 dependent pairs on or near the critical path. The two-pass fusing algorithm fuses more single-cycle ALU pairs on the critical path than a single-pass method does in [62] by observing that the criticality for ALU-ops is easier to model and that fused ALU-ops better match the 3-1 collapsed ALU units. The single-pass algorithm [62] would fuse the first `ADD` aggressively with the following store, which is typically not 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 the two-pass fusing algorithm comes with slightly higher translation overhead, its 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 fast, simple pipelines. For example, most x86 conditional branches are fused with the corresponding condition test instructions to dynamically form concise test-and-branches. This reduces much of the x86 condition code communication. The x86 ISA also has a limited number of general purpose registers (especially for the 32-bit x86) and the ISA is accumulator-based, that is, one register operand is both a source and destination. The consequent dependence graphs for micro-ops tend to be narrow and deep. This fact 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.

Finally, note that although the native x86 instruction set already contains what are essentially fused operations, the proposed fusing algorithm often fuses instruction pairs in different combinations than in the original x86 code, and it allows pairings of operation types that are not permitted by the x86 instruction set; for example the fusing of two ALU operations.

4.5 Code Scheduling: Grouping Dependent Instruction Pairs

The previous section concentrates on the kernel half of the macro-op fusing algorithm – the macro-op discovery algorithm that identifies appropriate dependent instruction pairs, and marks them to be fused into macro-ops. However, to complete the macro-op fusing algorithm, there is another half of the fusing algorithm: macro-op code scheduling that reorders instructions into dependent pairs.

Code scheduling algorithms in modern compilers typically re-order instructions to group *independent* instructions close together to enable better ILP. This kind of code scheduling proceeds by scheduling the exposed-set of an instruction dependency graph. However, the macro-op fusing algorithm needs to group *dependent* instruction pairs together. This cannot be achieved via conventional code scheduling manipulations.

After the macro-op discovery algorithm (the macro-op fusing algorithm discussed in the previous section – Section 4.4), data dependences have been considered and fusible pairs have been marked. Therefore the code scheduling algorithm considers the following. For a pair of identified fusible instructions, are there any other constraints or concerns that prevent moving the *middle instructions* (instructions between the head and the tail) to either before the head or after the tail? Such constraints could be memory ordering, interactions between different pairs, and any other special-case ordering concerns.

The code scheduling algorithm is listed in Figure 4.4. The major method, *DepCodeScheduler*, takes the original instruction sequence marked with macro-op fusing information as the input. It then processes the original sequence as an instruction (*uops* in the figure) stack and processes the *uop(s)* on the stack top. The output of the algorithm is a scheduled new sequence with *most* of the fusible pairs arranged consecutively and marked as macro-ops.

```
Algorithm: DepCodeScheduler (In: uops-seq, → Out: macro-op-seq)
1. foreach (uop in uops-seq, from the last to the first) {
2.    uops-stack.push(uop);
3. }
4. while (uops-stack is not empty) {
5.    uop = uops-stack.pop();
6.    if (uop is not fused) {
7.        Gen_mops (uop, mops_single → macro-op-seq);
8.    }
9.    else if (uop is head and the tail is consecutive){
10.      head = uop; tail = uops-stack.pop();
11.      Gen_mops (head, mops_head → macro-op-seq);
12.      Gen_mops (tail, mops_tail → macro-op-seq);
13.   }
14.   else {
15.      MidSet = uops-stack.pop(the set of uops between the head and tail);
16.      head = uop; tail = uops-stack.pop();
17.      can_fuse = partition(MidSet → PreSet, PostSet);
18.      if (can_fuse == true) {
19.          DepCodeScheduler  (PreSet → macro-op-seq);
20.          Gen_mops (head, mops_head → macro-op-seq);
21.          Gen_mops (tail, mops_tail → macro-op-seq);
22.          uops-stack.push(PostSet);
23.      }
24.      else {
25.          Gen_mops (uop, mops_single → macro-op-seq);
26.          uops-stack.push(MidSet + tail);
27.      }
28.   }
29.}
```

Figure 4.4 Code scheduling algorithm for grouping dependent instruction pairs

The algorithm operates in the following way.

A non-fused *uop* at the top of the stack is popped off the stack and generates a single instruction in the fusible ISA. The instruction is placed in the output buffer. This is shown from Line 6 to Line 8 in the algorithm.

A pair of consecutive and dependent instructions at the top of the stack that are marked for fusing are popped off the stack and generate two consecutive instructions in the fusible ISA. The head instruction has its fusible bit set to mark the two instructions as a fused macro-op (L9~L13).

If the *uop* at the top of the stack is part of a marked fusible pair, but the two instructions are separated by a set of middle instructions (called *MidSet*), the algorithm first pops off all the head, tail

and the *MidSet* instructions. Then the algorithm invokes a sub-algorithm to partition the *MidSet* into two sets. One is called *PreSet* that includes instructions that must be re-ordered before the marked macro-op head; the other is called the *PostSet* and it holds those instructions that can be re-ordered after the tail of the marked macro-op tail. If this partition fails due to memory ordering or other conditions, the head instruction is generated as a single instruction the fusing opportunity identified by the macro-op discovery algorithm is abandoned (Note: this re-ordering is done only for fusing dependent instruction pairs; the ordering between memory operations is strictly maintained as in the original x86 instruction sequence to maintain memory consistency model). Otherwise, if the partition succeeds, the scheduling algorithm recursively schedules the *PreSet* first, and then generates the macro-op pair being processed. Finally it pushes all the *PostSet* instructions back to the stack for further processing (L14 ~ L28).

The sub-algorithm, *partition*, is designed to honor all the original dependences in the original x86 code sequences, including memory operation ordering. The partition sub-algorithm prefers to assign instructions to *PostSet* for future processing. *PreSet* only holds instructions that must be scheduled before the head to maintain the correctness of the re-ordering.

4.6 Simple Emulation: Basic Block Translation

As illustrated in Chapter 3, for most workloads, hotspots constitute only a small fraction of all executed static instructions. If the full-blown SBT procedure described above is applied to all the static instructions executed, the runtime overhead would be very significant. Therefore, like many other adaptive translation systems, the *x86vm* DBT software employs the staged translation strategy as discussed in Chapter 2 and 3. When instructions are first encountered, the VM uses fast and simple basic block translation (BBT). BBT improves performance over interpretation by (a) reducing the x86 instruction fetch/decode overhead to only one time per basic block; (b) exploiting native register

mapping of the x86 architected state as well as specializing the native emulation code for each x86 instruction; (c) facilitating hardware accelerators to speed up the BBT translation. With these objectives, BBT is more efficient than interpretation as later results demonstrate. Because no optimization is performed, a basic block is selected as the translation unit to reduce code management complexity and avoid tail replication for translation units beyond a basic block.

The basic block translator fetches and decodes x86 instructions one-by-one. After each x86 instruction is discovered, BBT cracks it into multiple fusible ISA instruction(s) and stores the translation into the BBT code cache. The register mapping convention in Section 4.3 is enforced to communicate with other translations without overhead. Conceptually, the functionality of this BBT translation is very similar to the hardware decoders at the pipeline front-end of contemporary x86 processors [37, 51, 53, 58, 74].

However, there are several DBT specific concerns. The first DBT-specific issue regards branches. Each x86 branch instruction is translated into a corresponding fusible ISA branch instruction. If the branch target is known and the target translation is available, the fusible ISA branch transfers control directly to the target translation. Otherwise, the fusible ISA branch transfers control back to the VMM runtime, which will patch the branch and link directly to the target translation that will be available at the VMM link time. This translation control transfer mechanism is the same as that of the superblock translation.

The second DBT-specific regards hotspot profiling. In many VM systems, including the IBM Daisy and Transmeta co-designed VM systems, the initial emulation system also performs profiling and this adds extra overhead. The *x86vm* framework features special hardware support to detect hotspots. The goal is to reduce runtime overhead and the BBT translation complexity. As will be

seen in Chapter 5, the removed BBT profiling will also facilitate hardware acceleration of the BBT translation process.

The third DBT-specific issue is code caching for BBT translations. Like optimized hotspot translations, BBT translations are cached in a BBT code cache for reuse. This cache space may require extra memory space. Typically, BBT translations are not repeatedly reused over a long period, however, so flushing a limited BBT code cache is acceptable, especially if BBT can re-translate very quickly.

4.7 Evaluation of Dynamic Binary Translation

There are two complementary aspects to the dynamic binary translation system: the effectiveness of its macro-op fusing algorithms and the efficiency of its translation process.

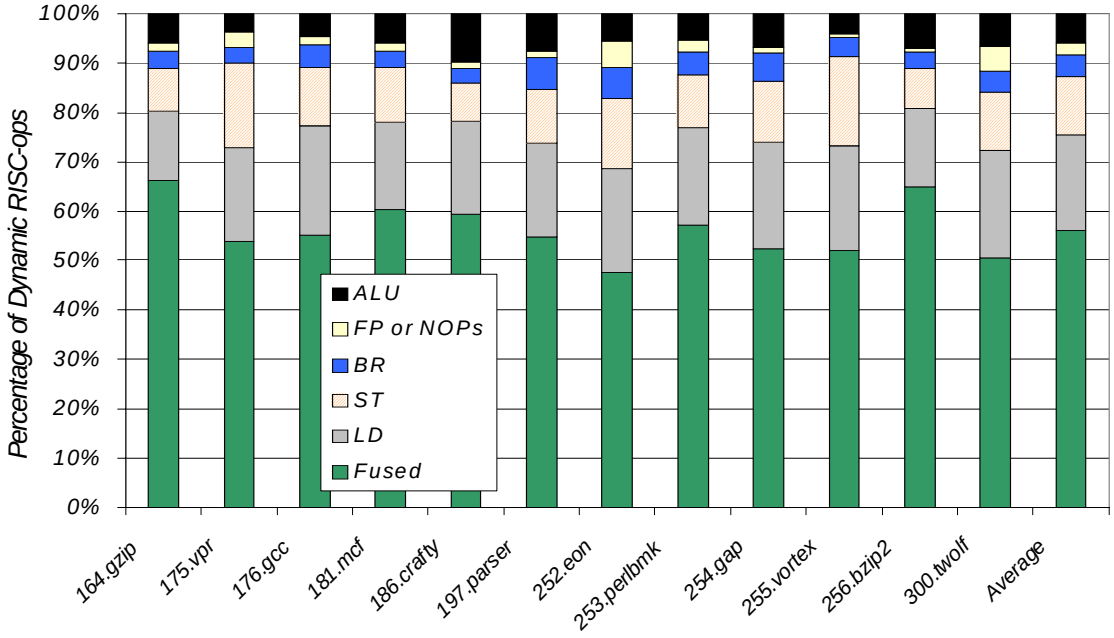
Evaluation of the effectiveness of macro-op fusing algorithms

A primary functionality of the dynamic binary translator is the fusing of macro-ops. The degree of fusing, i.e., the percentage of micro-ops that are fused into macro-ops, determines how effectively the *macro-op execution* engine can utilize the pipeline bandwidth. Additionally, the profile of non-fused micro-ops implies how the pipelined scheduler affects IPC performance.

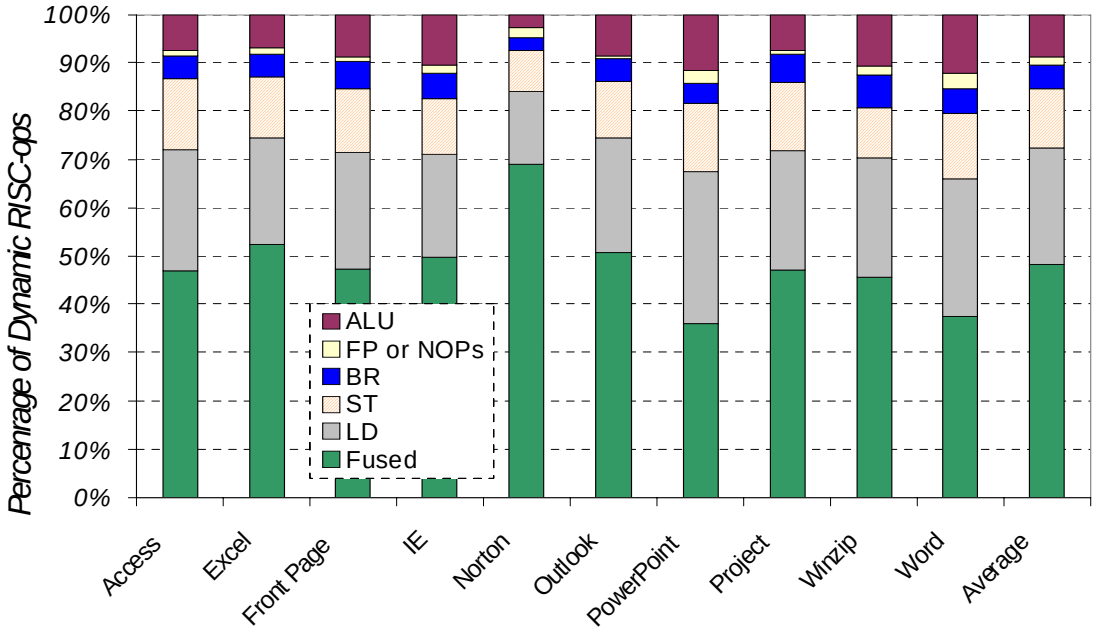
Figure 4.5 plots fusing profiles. The x-axis shows the individual benchmarks. The y-axis shows the percentages of dynamic micro-ops that are fused, and, if not fused, they are further classified into the categories of loads (LD), stores (ST), branches (BR), floating point (FP) or NOP and ALU-ops. Clearly, the macro-op fusing profile is fairly consistent across all the SPEC2000 integer benchmarks and the Windows workloads tested.

Figure 4.5a presents the results for the SPEC2000 integer benchmarks. On average, more than 56% of all dynamic micro-ops are fused into macro-ops, much higher than the sub-40% coverage achieved in hardware-based fusing reported in the related work [20, 27, 81, 110] for the common SPEC2000 integer benchmarks. Non-fused operations are mostly memory LD/ST operations, branches, floating point operations and NOPs. Non-fused single-cycle ALU micro-ops are only 6% of the total micro-ops, thus greatly reducing the penalty due to pipelining the macro-op scheduler.

For the WinStone2004 Business Suites, more than 48% of all dynamic micro-ops are fused into macro-ops (Figure 4.5b). Non-fused single-cycle ALU micro-ops are about 8% of all dynamic micro-ops. It is clear that the Windows application workloads not only have relatively larger instruction footprints, but also are more challenging for macro-op fusing.



(a) SPEC 2000 integer



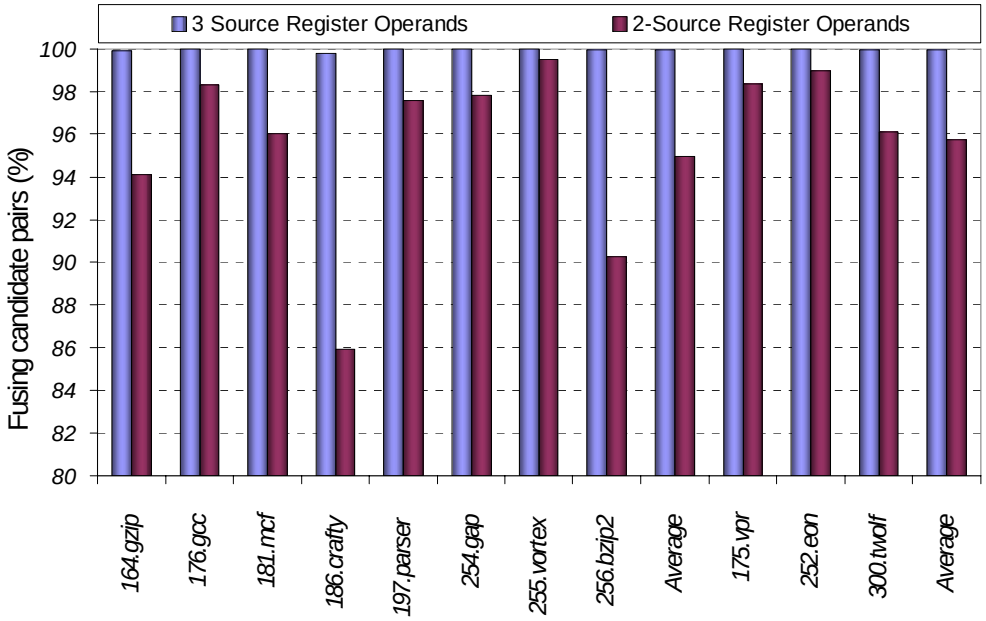
(b) WinStone2004 Business Suites

Figure 4.5 Macro-op Fusing Profile

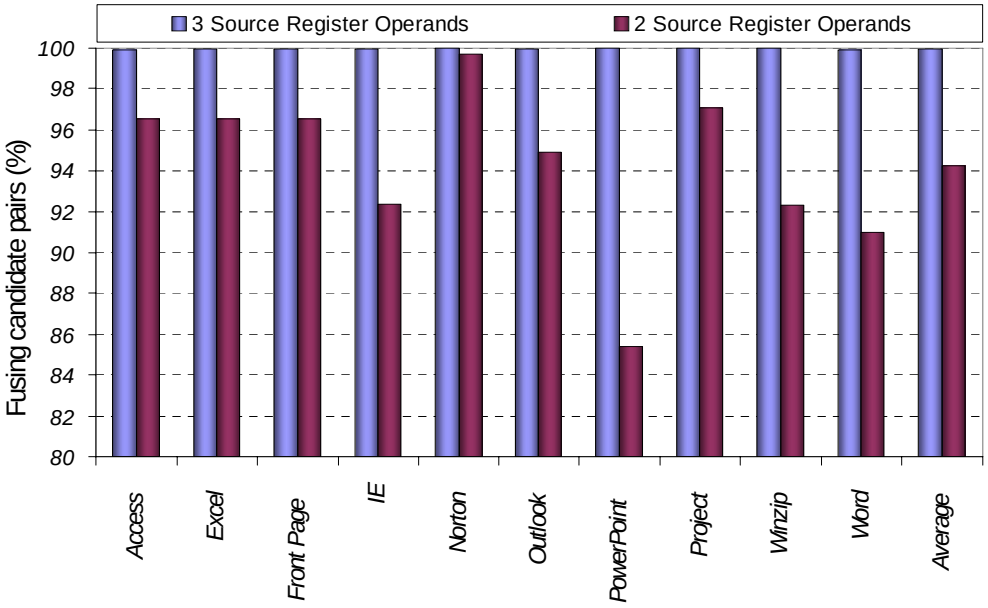
The micro-ops that are fused into macro-ops (nearly 50% to 60%) lead to an effective 25% to 30% bandwidth reduction throughout the pipeline. These fusing rates are lower than a single-pass fusing algorithm [62] which can fuse (65% for SPEC2000 integer and 50+% for WinStone2004). However, the improved two-pass fusing algorithm prioritizes critical single-cycle ALU-ops for fusing. Preliminary performance experiments with the single-pass fusing algorithm actually show inferior IPC performance numbers because its greedy fusing heuristic does not prioritize critical dependences and single-cycle integer ALU operations. Results in Chapter 6 will show the superior performance enabled by the two-pass macro-op fusing algorithm.

In theory, pairing arbitrary RISC-style micro-ops together may lead to macro-ops having three or more source register operands because each micro-op can have three register operands and two of them can be source operands. More source register-operands would require each macro-op issue queue slot to accommodate more source register specifiers than a conventional issue queue slot. To avoid this and honor the heuristic for reduced pipeline complexity, our fusing algorithms do not fuse micro-op pairs that have more than two distinct source register operands. However, it is interesting to evaluate whether this heuristic passes up a significant number fusing opportunities.

Figure 4.6 shows the percentages of fusing candidates that fall into macro-op categories with at most two source register operands, and at most three register operands. It is clear that almost all fusible candidate pairs have three or fewer input register specifiers for both SPEC2000 integer and WinStone2004 Business Suite workloads. For SPEC2000 integer, nearly 96% of all fusible candidates have two or fewer distinct source registers. For WinStone2004 Business, this percentage is a little more than 94%. In both benchmarks, the worse case has 85% of all fusible candidate pairs with two or fewer distinct source register operands. Overall, we conclude that the two-operand heuristic does not appear to lose many fusing opportunities.



(a) SPEC 2000 integer



(b) WinStone2004 Business Suite

Figure 4.6 Fusing Candidate Pairs Profile (Number of Source Operands)

After fusing, the non-fused micro-op profile provides important information. In particular, only 6~8% non-fused micro-ops are single-cycle ALU operations that produce a value consumed by dependent micro-ops. Other non-fused micro-ops are either multi-cycle operations, e.g. LD and FP ops, or micro-ops that do not generate a value, e.g. branches and NOPs. Therefore, the pipeline for macro-op execution can be designed as though all operations take two or more cycles for execution. For example, the critical macro-op scheduler/issue logic can be pipelined and the execution stage can be simplified.

The profile of the fused macro-ops also sheds some light on potential gains for performance or efficiency. Figure 4.7 shows that the fused macro-op pairs fall into the following three categories:

- *ALU-ALU macro-ops*. Both the head and tail micro-ops are single-cycle ALU operations. The head produces a value that is consumed by its tail and this value communication is fused. The fused ALU pairs must meet the constraints of a collapsed 3-1 ALU [98, 106].
- *ALU-BR macro-ops*. The head is a single-cycle ALU operation. The tail is a branch operation. The head produces a value consumed by the tail. In most cases, the head is a condition code set ALU-op and the tail is a conditional branch based on the conditional code.
- *ALU-MEM macro-ops*. The head is a single-cycle ALU operation that produces a value for its tail that is memory operation, either a LD or a ST. The head ALU is an address calculation operation in most cases and in some cases produces a value to be stored to memory.

Figure 4.7 shows that for SPEC2000 benchmarks, 52% of fused macro-ops are ALU-ALU pairs, 30% pairs are ALU-BR, and only 18% of total macro-ops are ALU-MEM pairs. For Windows workloads, the results are slightly different. 43% of macro-ops are ALU-ALU pairs, 35% of macro-ops are ALU-BR pairs and 22% macro-ops are ALU-MEM pairs.

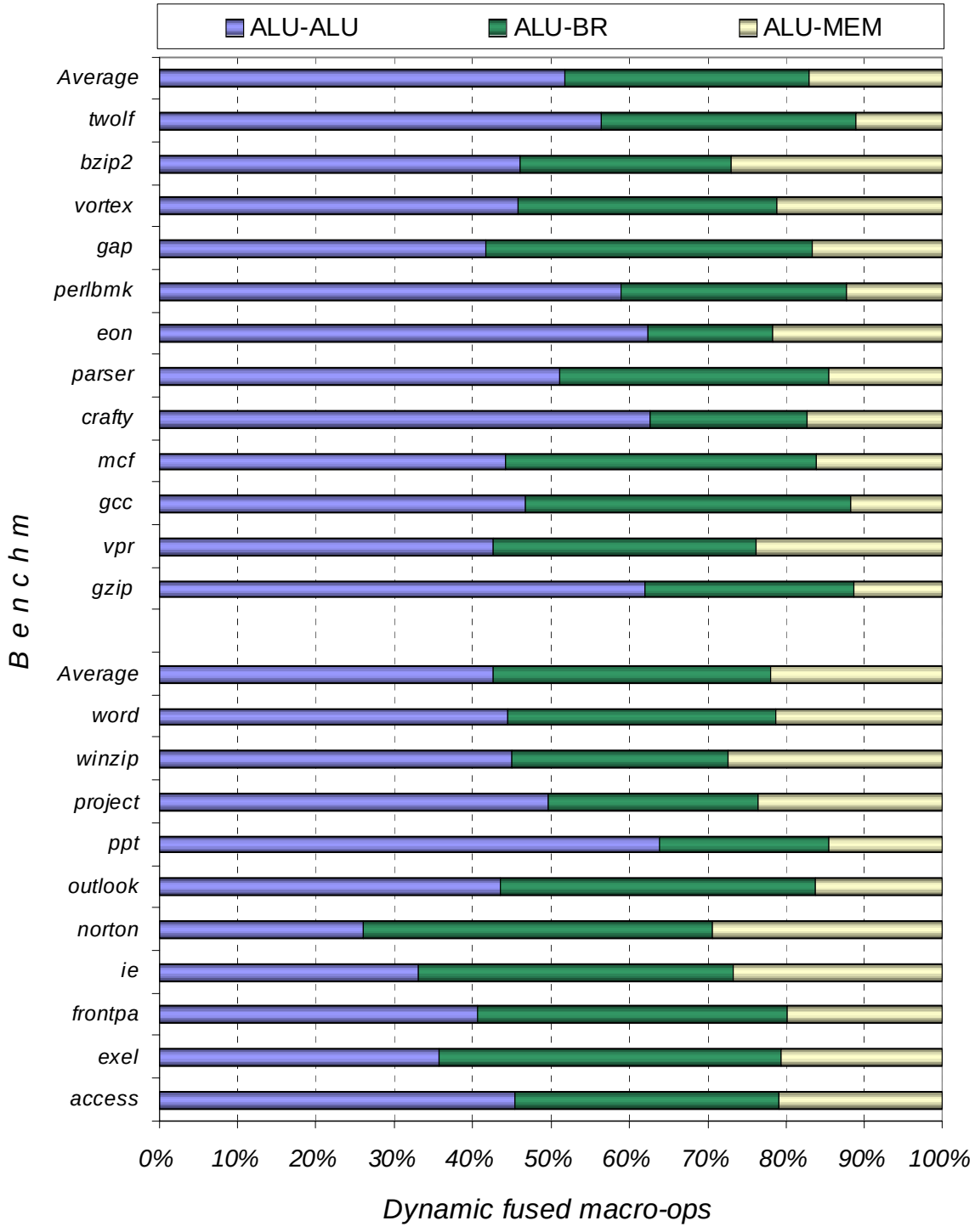


Figure 4.7 Fused Macro-ops Profile

The fusing heuristic that targets criticality suggests fusing nearby dependent instruction pairs. Figure 4.8 shows the distance distribution of the fused pairs of micro-ops. The x-axis (vertical) lists the tested benchmarks. The y-axis (horizontal) shows the percentage of dynamic macro-ops that fuse two micro-ops with a certain distance in the original micro-op sequence (cracked from the x86 instructions). The distance is measured as the instruction ordering/index difference in the original micro-op sequences. For example, distance "1" means consecutive instructions in the original micro-op sequence and distance "2" means there is an instruction in the middle between the head and tail in the original sequence.

Figure 4.8 illustrates that about 65% of all dynamic macro-ops are fused from two consecutive micro-ops in the original sequence, though not necessarily from the same x86 instruction. About 20% of the macro-ops are fused from micro-op pairs separated by distance 2. About 10% of the macro-ops are fused from micro-op pairs separated by 3 or 4. Less than 5% of all dynamic macro-ops are fused from micro-op pairs that are farther than 5 instructions apart.

Figure 4.8 suggests that a hardware macro-op fusing module might be feasible for targeting consecutive micro-op pairs that are suitable for fusing. However, this is not as straightforward as it first appears. Such a fusing method would likely function like the single-pass fusing algorithm [62] that fuses so greedily that it can actually hurt performance. A hardware fusing module also brings extra pipeline complexity. This opportunity is left as a topic for future research.

Considering the data from both Figure 4.7 and 4.8 together, it is evident that fused ALU-op pairs and dynamically synthesized powerful branches (a fused condition test with a conditional branch) will provide the primary performance boost. The accelerated address calculations will also be a significant performance contributor. Most macro-ops are fused within the same basic block and will therefore cause less complications for consistent machine state mapping.

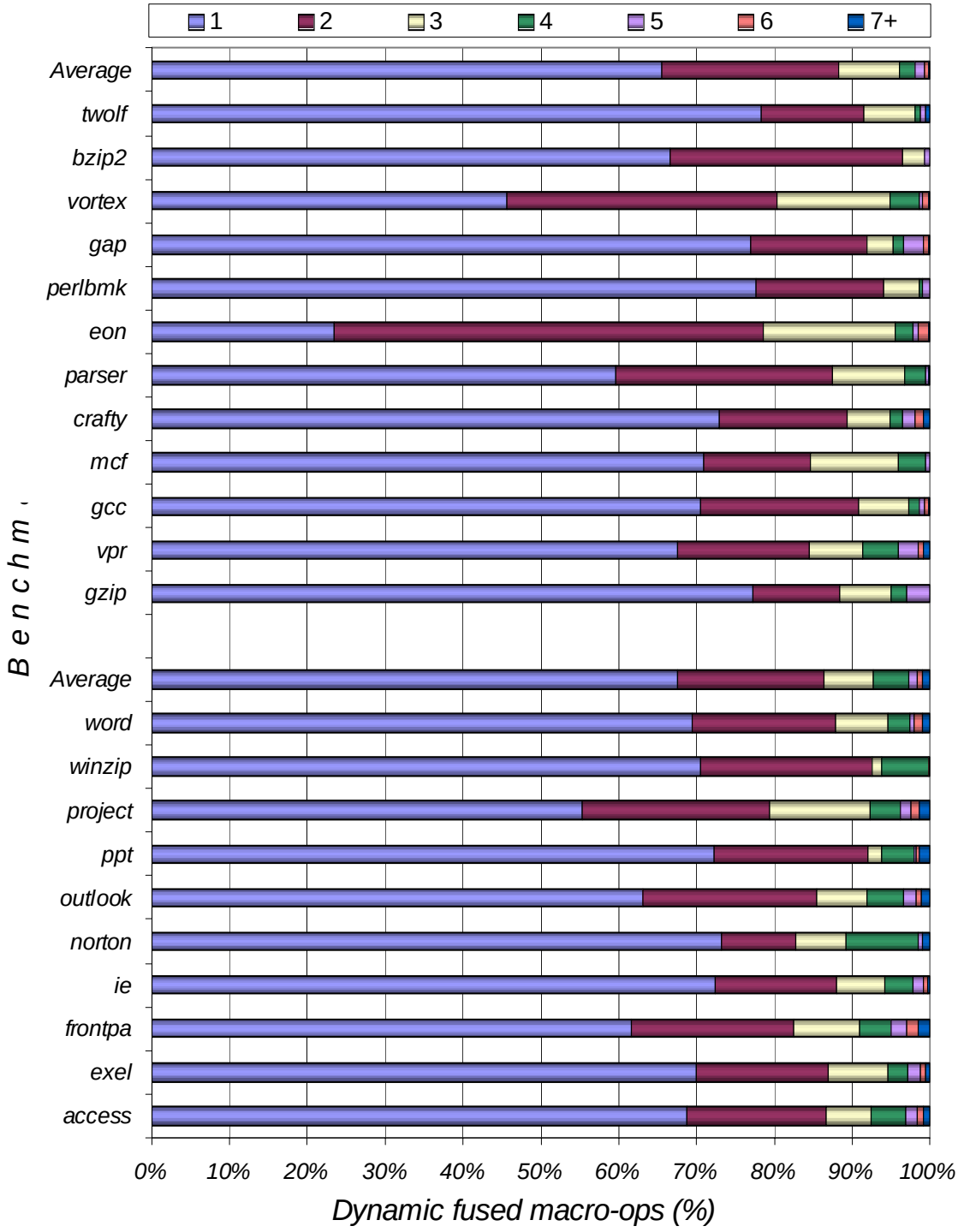


Figure 4.8 Macro-op Fusing Distance Profile

As a side effect of dynamic binary translation, code straightening effects caused by superblock formation are also of interest. Our data suggests that a dynamic superblock in our VM system typically consists of three to five basic blocks. Many unconditional direct jumps are removed for improved instruction fetching. Results to be given later (Chapter 6, Section 6.3) will demonstrate the performance effects.

Evaluation of efficiency of the dynamic binary translation software

As the performance models in Chapter 3 suggest, translation overhead in general is the product of the code cache misses M_{DBT} and the code cache miss penalty Δ_{DBT}. And in our two-stage translation system, it is caused by both BBT and SBT. The profile information for BBT and SBT is then needed to evaluate the efficiency of the translation system. This information will also help to develop hardware assists for DBT system. Since the M_{BBT} and M_{SBT} are essentially characteristics of programs, here we focus on profiling the Δ_{BBT} and Δ_{SBT} components instead.

Figures 4.9 and 4.10 show the runtime overhead profiles for the Δ_{BBT} and Δ_{SBT} components. In both the figures, the x-axis lists the benchmarks. The y-axis shows the number of dynamic translator instructions to translate each x86 instruction. The BBT is accurately modeled via its native assembly code in our *x86vm* infrastructure and its overhead is measured in terms of number of fusible ISA instructions. On the other hand, the SBT is too complex to be modeled in the same way. Hence, we use profiling tools to profile the SBT that is written in C++ and runs as x86 binary. The overhead is then measured in terms of x86 instructions, and can be converted to fusible ISA instructions.

For each x86 instruction, the BBT overhead can be modeled as, $\Delta_{BBT} = T_{decode} + T_{encode}$. The decode part includes determining the x86 instruction boundary, decoding, and indexing to the cracking routine for the x86 instruction. The encode part includes cracking that x86 instruction and encoding the fusible ISA micro-ops. Moreover, there are other miscellaneous VM runtime overheads

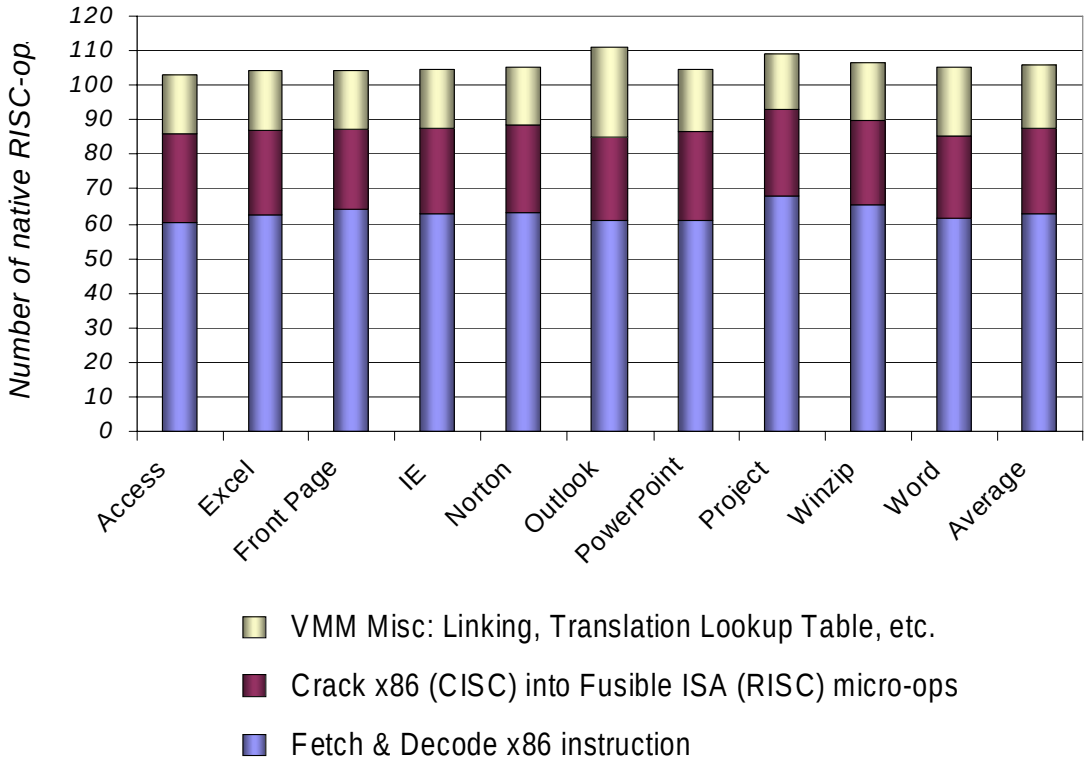


Figure 4.9 BBT Translation Overhead Breakdown

spent for branch instruction linking/patching, translation registration, translation table lookup and code cache management etc.

Figure 4.9 indicates that on average, the software runtime overhead for each BBT translated x86 instruction is about 106 native fusible ISA instructions. Figure 4.9 further breaks down the BBT runtime overhead into: fetch/decode (T_{decode}), cracking x86 instructions (T_{encode}) and other miscellaneous tasks (T_{misc}). Because of the complexity of the x86 instruction set, our BBT has fairly heavy overhead even for the BBT written in fusible ISA assembly. There should be space for improvement. However, it is unlikely to be significant.

For each x86 instruction, the SBT translation and optimization overhead can be modeled as, $\Delta_{BBT} = T_{decode} + T_{optimize} + T_{encode}$. Here, $T_{optimize}$ is the dominant part for translation and optimization.

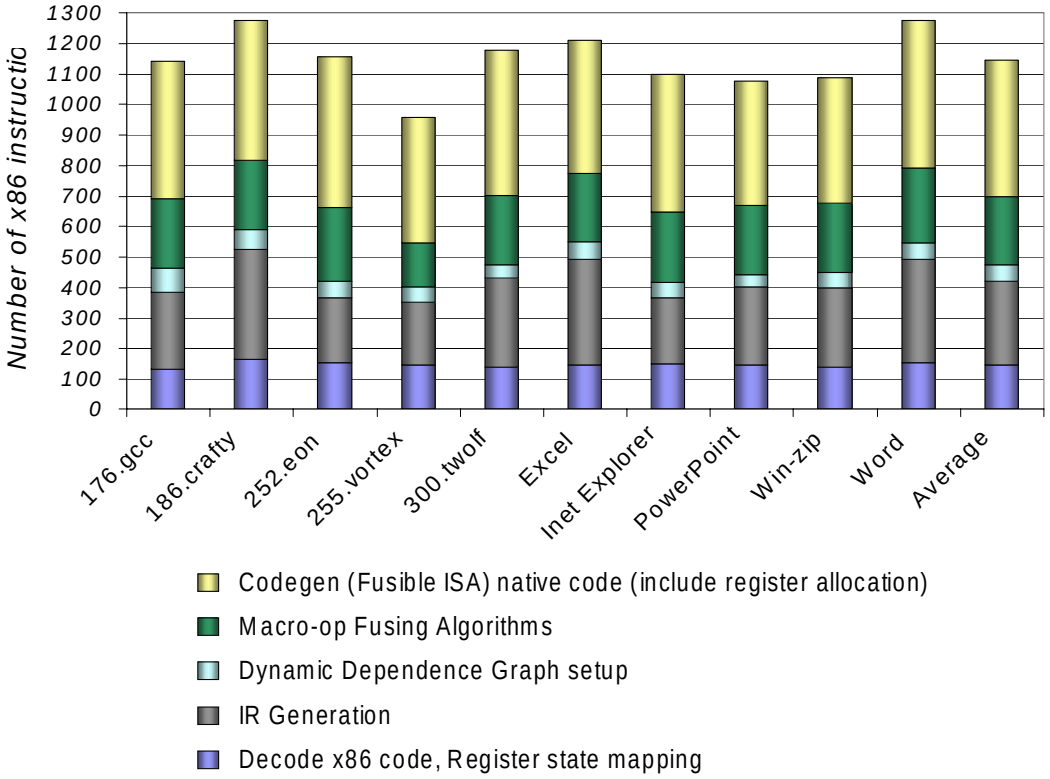


Figure 4.10 Hotspot (SBT) Translation Overhead Breakdown

The SBT in our VM system is written in C++, not in fusible ISA assembly. Therefore, it is profiled with the Intel *VTune* for Linux 2.0 and GNU *gprof,* rather than detailed simulations as for BBT. The overhead is broken down by C++ functions, rather than the above model. For example, the *Codegen* bars in Figure 4.10 also include the overhead for grouping dependent instruction pairs together and register allocation for state mapping.

Figure 4.10 indicates that per x86 instruction SBT overhead is about 1150 x86 instructions. There is no dominant function for the SBT overhead. And there are other considerations for reducing SBT overhead. For example, the hotspot translation and optimization should be flexible and capable to target its co-designed processor, and a balanced VM system applies hotspot optimizations only to a small footprint of static code.

4.8 Related Work on Binary Translation Software

There are many research efforts and commercial products related to binary translation software. In this chapter, we emphasized the major issues regarding translation and optimization of x86 binary code for our co-designed macro-op execution engine. There are some details that have not been specifically addressed, such as self-modifying code and page reference bookkeeping for the architected ISA (x86). These issues have already been tackled in related work. Translation and optimization for other architecture innovations such as VLIW ISAs are also considered as related work.

The original version of IBM DAISY [41] used a single-stage translation system that translated all PowerPC (architected ISA) instructions in a given physical page when an untranslated code page was executed . This pioneering work addressed the issues of precise exceptions, and page and address mapping mechanisms to maintain 100% architecture compatibility. Later versions of IBM DAISY [3, 42] adopted an adaptive/staged translation strategy, i.e. first interpret PowerPC instructions before they reach certain threshold. Once the hot threshold is reached, a hotspot is identified and a translation unit, tree region/group, is formed, followed by optimizing translation.

The Transmeta Crusoe processor [54, 82] also uses staged emulation with a software interpreter first emulating x86 code, followed by translation and optimization of hotspots. The initial interpretation process also performs software online profiling. Each code region is interpreted multiple times up to a given threshold, and then translation to native VLIW code is performed. The Crusoe Code Morphing Software [36] systematically addressed the issue of x86 self-modifying and self-referencing code, which are quite frequent in x86 device drivers, legacy software, and security related software. The CMS system features specific approaches for different scenarios to maintain efficiency and 100% compatibility.

The Transmeta Efficeon processor emloys a unique 4-stage translation strategy (including the initial interpretation [83]) to reach the right level of optimization for different code regions. The translation unit is a tree region.

All the discussed IBM and Transmeta co-designed virtual machines happened to use VLIW engines as the co-designed microarchitecture. With the in-order VLIW approach, considerable software optimizations are required for scheduling/reordering instructions, especially if speculation is implemented via the instruction set [70]. The optimization overhead for VLIW engines are quite heavy, 4000+ native operation per PowerPC [41]. The data for Transmeta CMS is estimated to be similar. The Transmeta processors provide checkpoint and fenced store-buffer mechanisms to support such reordering.

Of course, the processor for a co-designed VM does not necessarily have to be a VLIW. The co-designed ILDP (Instruction Level Distributed Processing) VM [76, 78] explored translation algorithms for its superscalar-like processor. In that approach, the DBT software forms strands (chains of dependent instructions) for a co-designed ILDP microarchitecture. As with our macro-op execution engine, the ILDP processor microarchitecture is fully capable of dynamic instruction scheduling. For such out-of-order architectures, the optimization software is anticipated to be significantly simpler than that for the VLIW implementations.

Besides those that use the co-designed VM paradigm, there are other VM systems that emulate program binaries. These systems run software distributed in *source* ISA on top of *target* ISA platforms. The interface these VM systems handle is ABI (Application Binary Interface), which includes *user-mode* part of the source and target ISA(s), OS system calls and certain exceptions.

The Intel IA-32 EL [15], for example, dynamically translates x86 instructions into IA-64 [70] VLIW instructions on-the-fly for user-mode code only. IA-32 EL is a two-stage translation system

that does not interpret. All x86 code is translated (when first executed) with a simple and fast BBT translator. The BBT translator instruments its translations to collect execution profiles. The BBT translator also applies certain rules when generating code so that the precise state can be recovered easily should an exception occurs anywhere within the basic block translation. For example, for each x86 instruction, no x86 architected state is modified until all operations performed by the instruction finish without exception. Later, after hot code is detected, a heavy-duty optimizing translator is applied to generate optimized code for the target Intel IPF processors.

The DEC FX!32 [24, 60] implements a high performance x86 interpreter and a profile-guided *static* binary translator for running the x86 Windows applications on DEC Alpha Windows platforms. The interpretation overhead is rather low for the x86 – less than 50 Alpha instructions per x86 instruction. To strive for performance, FX!32 does not maintain intrinsic binary compatibility. For example, it does not emulate x87 floating point faithfully, and it cannot materialize the precise x86 state at an arbitrary point within its translation.

There are also binary translators for RISC ISAs, for example, PA-RISC to IA-64 [130], VEST/ TIE for Alpha and *mx* for MIPS [113]. RISC instruction sets have much lower software decode and interpretation overhead when compared with a CISC instruction set such as the x86.

Dynamo [13] is a PA-RISC dynamic optimization system. It interprets first and optimizes hot-spots detected. It bails out if its optimization does not improve performance. DynamoRIO [22] is a framework for runtime code transformation and inspection for the x86.

Table 4.1 Comparison of Dynamic Binary Translation Systems

DYNAMIC BINARY TRANSLATION SYSTEMS	TRANSLATION OBJECTIVES	TRANSLATION STRATEGY	COLD CODE	HOTSPOT DETECTION	HOTSPOT OPTIMIZATION
UQDBT	Multi-source/target Translation Study	Adaptive 2-stage Software	BBT	Software Edge Profiling	Generic Hot Path Opts.
Shade	Simulation via Binary translation	Always Translate w/ Simulation functions	n/a	n/a	n/a
FX!32	Run Windows x86 apps on Alpha	Online Interpreter, Offline Translation	Interpreter	Software (Interpreter)	Static Translator Optimize for Alpha
IA-32 EL	Run IA-32 apps on Intel IPF platform	Adaptive 2-stage Software	BBT	Software Instrumentation	Optimize for IPF processors
x86vm	ISA mapping for efficiency, performance	Adaptive 2-stage, HW/SW Co-Designed	Dual mode / BBT	Hardware Detector	SW DBT w/ Simple HW Assist. Fuse macro-ops
CMS: Crusoe	ISA mapping for low power and HW complexity	Adaptive, 3-stage Software	Interpreter and BBT	Software	Tree-region Opts for its 4-wide VLIW engine
CMS: Efficeon	ISA mapping for Low power and HW complexity	Adaptive 4-stage SW w/ some HW support	Interpreter and BBT	Software?	Tree-region Opts for its 8-wide VLIW engine
DAISY / BOA	ISA mapping for Performance (ILP)	Adaptive 2-stage Software	Interpreter	HW counters / SW Instrumentation at group exits	Tree group Opts for VLIW engines
Dynamo (RIO)	Dynamic code Opts and inspection	Adaptive 2-stage Software	Interpreter. RIO: Native EXE or BBT	Software MRET hot path formation	Dynamo: Opts RIO: flexible for multi-purpose
Jikes RVM	HLL Java Program Platform Independence	Adaptive 2-stage Software	Simple JIT	Software	Optimizing JIT
Microsoft CLR	Multi-Language Platform Independence	Adaptive 2-stage Software	Simple JIT	Software profiling Instrumentation	Optimizing JIT

As a brief summary of the related work on DBT, Table 4.1 compares many existing dynamic binary translation systems. Note that all systems perform translation chaining inside code caches for direct branches. The ILDP VM also uses hardware support to chain indirect branches.

References [4, 5] provide excellent surveys regarding the state-of-the-art of dynamic binary translation and important issues such as precise exception handling. Le [85] shows how to extend register live range to support precise exception handling for out-of-order scheduling in DBT.

Chapter 5

Hardware Accelerators for x86 Binary Translation

As discussed in Chapter 3, the *x86vm* translation strategy includes new primitives and assists to accelerate the critical part of the VM runtime software, especially for BBT translation and hotspot detection. In this Chapter, we propose two new hardware accelerators for BBT, one at the pipeline front-end (Section 5.1) and the other at the pipeline backend (Section 5.2). Hardware assists for hotspot detection and profiling are described in related work. In Section 5.3, we discuss how these assists help co-designed VM systems in particular. The proposed hardware assists are evaluated in Section 5.4. Section 5.5 surveys related work.

5.1 Dual-mode x86 Decoder

In conventional x86 processor designs, x86 instructions are decoded into RISC-style operations called micro-ops/uops. Although this hardware translation cannot achieve the powerful translation and optimizations we propose for hotspot translation, this mechanism can perform the simple translation sufficient for program startup phases.

In contrast, most start-of-the-art co-designed VM systems run a slow software interpreter or simple BBT translator to get beyond program startup phases and then count on optimized hotspot code to compensate for the startup performance loss. The extra software runtime overhead may not always be paid back (at least for some of the scenarios discussed in Chapter 1).

We propose a method for collecting the advantages of both types of systems: the solid startup performance of conventional x86 processors, and the flexible advanced software translation for hotspot performance optimization. The key is a mechanism that seamlessly combines these two execution modes (dual mode) together.

The key to the method lies in the decoder stage. For a macro-op execution pipeline, fusible ISA instructions can be decoded by simple RISC-style decoders. However, for x86 (CISC) processor pipelines, the x86 instructions go through a two-level decoding process. The first level decoder identifies x86 instruction boundaries and cracks x86 instructions into “vertical” RISC-style micro-ops [119]. Then, a second level decoder generates the “horizontal” decoded signals and controls used by the backend pipeline. The second level decoder is in fact very similar to the RISC-style decoders in our macro-op execution microarchitecture. A two-level decoder is especially suited to a CISC ISA because complex CISC instructions must both be decomposed (*cracked*) into RISC-style micro-ops and be decoded into pipeline control signals.

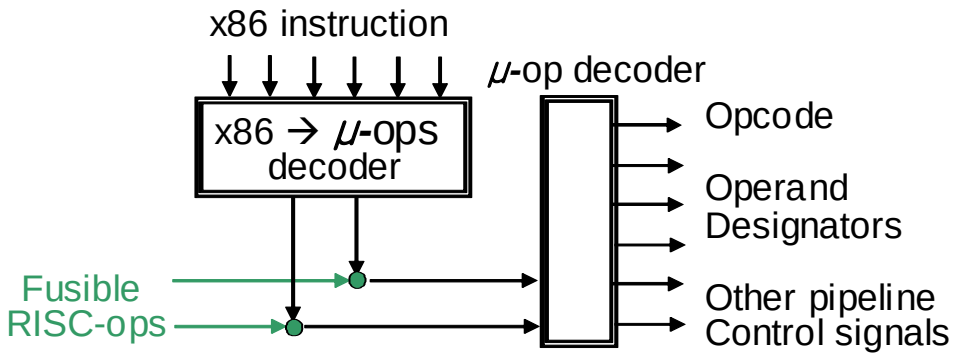


Figure 5.1 Dual mode x86 decoder

The two-level decoder was first published for the microcode engine used in the Motorola 68000 and has been deployed by most modern CISC processors. Our macro-op pipeline leverages this two-level decoding approach and employs a dual mode (two-level) decoder (Figure 5.1) that targets CISC ISA(s) in particular. The first level of the dual mode decoder identifies x86 instruction boundaries and cracks x86 instructions into "vertical" RISC style micro-ops. However, the dual mode aspect dictates that the RISC micro-ops are in the same 16-bit/32-bit micro-op format as for the fusible ISA. The second level of dual mode decoder then generates conventional "horizontal" decoded control signals. To this structure, we add a bypass path (Figure 5.1) around the first level decoder, which enables the decoder to be used in dual modes (x86 and fusible ISA).

The two modes of the dual-mode decoder are named *x86-mode* and *native mode*. In *x86-mode*, x86 instructions are fetched from memory, and both decode levels are used. In *native-mode*, hotspot translated implementation instructions are fetched from the code cache. These fusible ISA instructions bypass the first level decoder and only go through the second level decoder. With the dual-mode decoders, both architected ISA (x86) code and implementation ISA instructions can be processed by the same pipeline. The ability to support x86 mode eliminates the need for BBT, along with its translation overhead and any side effects on the memory hierarchy.

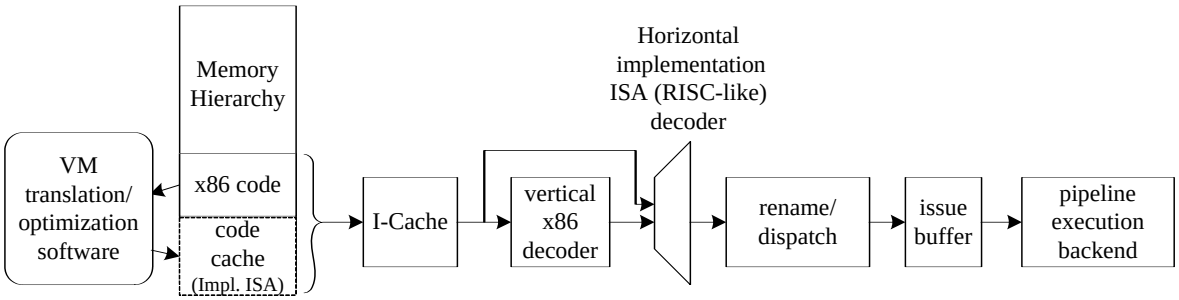


Figure 5.2 Dual mode x86 decoders in a superscalar pipeline

As the processor runs, it switches back and forth between x86-mode and native-mode, under the control of VMM software. The x86-mode is entered if a piece of x86 code has no translation in the code cache; this is done when a program starts up, for example. As the program runs, some parts become hotspot. Once a hotspot has been translated and optimized, the VMM software switches to native mode to take advantage of its efficiency and performance.

When executing in x86-mode, instructions pass through both decode levels (Figure 5.2). In this case, the dual-mode decoders generate micro-ops with a code quality similar to conventional x86 decoders. Furthermore, the macro-op execution pipeline is an enhanced superscalar that processes non-fused micro-ops in a similar way as a conventional superscalar, except the scheduler is pipelined. The pipelined scheduler loses the back-to-back issue capability for dependent micro-ops that are not fused; however, it also leads to higher clock speeds. Therefore, in x86-mode, performance will be similar to a conventional superscalar x86 processor.

When executing in native mode, fused RISC-style micro-ops pass through only the second level of the decoder (Figure 5.2), leading to a shorter pipeline frond-end for branch misprediction penalty. The complex first level of the dual mode decoder can be turned off.

A side effect of using the dual-mode approach is that profiling software cannot be embedded into BBT generated code -- because there is no BBT code. As a consequence, the design should

employ profiling hardware similar to that used by Merten *et al*. [96, 98]. This hardware's sole function is to detect hotspots. When a hotspot is detected, the hardware invokes the VMM software which can then organize hotspot code into superblock(s), translate and optimize it, and place the optimized superblock(s) in the code cache.

Dual mode decoders are fast and fit well in a conventional superscalar design. The replacement of a single-level decode table with a two-level decoder represents a good hardware tradeoff which results in fewer transistors, as explained by the Motorola 68000 designers [119]. This approach, when extended to dual mode operation, adds relatively little extra hardware to a conventional two-level CISC decoder implementation -- the bypass path around the first level decoder.

5.2 A Decoder Functional Unit

The front-end dual mode decoder described in the previous subsection modifies a critical part of the processor pipeline and must be able to decode instructions at full bandwidth. Furthermore, it must be designed to implement the complete architected ISA (x86). However, dynamic binary translation software may fit better with flexible, programmable and more complexity-effective hardware assists.

An alternative approach is to implement the hardware assist in the form of a programmable functional unit at the pipeline back-end. A functional unit at the execution stage is less intrusive than the front-end dual mode decoder, does not need to provide the high bandwidth of the front-end decoders, and can target only the common cases, not all cases.

As previously illustrated, during initial emulation, BBT introduces the major runtime overhead, and the dominant part of BBT is to decode and crack x86 instructions into micro-ops. According to our measurements, on average, about 90 out of the 106 μops overhead for translating each x86

instruction in our BBT system is related to instruction decoding and cracking. Therefore, a functional unit that performs these operations should significantly speed up BBT.

Table 5.1 Hardware Accelerator: XLTx86

NEW INSTRUCTION	BRIEF DESCRIPTION
XLTx86 Fsrc, Fdst	Decode an x86 instruction aligned at the beginning of the 128-bit Fsrc register, and generate RISC-style 16b/32b uops into the Fdst register. This instruction affects the CSR status register

We propose such a backend functional unit that is accessed through a new instruction in the implementation ISA. Table 5.1 briefly describes the new instruction: XLTx86. XLTx86 accesses the 128-bit F registers that are architected for mapping the x86 FP/media states. Additionally, XLTx86 operates on a special flag status register CSR that is explained below.

```
0.   HAloop:
1.   LD     Fsrc, [Rx86pc]
2.   XLTx86 Fdst,  Fsrc
3.   Jcpx   complex_x86code
4.   Jcti   branch_handler
5.   ST     Fdst, [Rcode$]
6.   MOV    Rt0, CSR
7.   AND    Rt1, Rt0, 0x0f  ::  ADD   Rx86pc, Rt1
8.   AND.x  Rt2, Rt0, 0xf0  ::  ADD   Rcode$, Rt2
9.   JMP    HAloop
```

(a). Code for the HW assisted fast BBT loop

Flag_cti	***Flag_cmplx***	***µops_bytes (4-bit)***	***x86_ilen (4-bit)***

(b). The control and status register (CSR) format for XLTx86

Figure 5.3 HW accelerated basic block translator kernel loop

Figure 5.3a illustrates the kernel loop used by the VMM for hardware accelerated BBT (in the implementation ISA assembly language). Rx86pc is an implementation register that holds the architected x86 PC value; this register points to an instruction in the x86 instruction memory.

To be more specific, x86 instructions are fetched by a load operation into register Fsrc. Because x86 instructions are from one byte to seventeen bytes long (and very few are more than eleven bytes in real code), the Fsrc register holds at least one x86-instruction. The fetched x86 instruction is aligned at the beginning of the Fsrc register. The next instruction, XLTx86, then decodes and cracks the x86-instruction into uop(s). The input to XLTx86 is the Fsrc register. The output uops are placed in the Fdst register and flags are set in the CSR status register. The format of the flag status register CSR is shown in Figure 5.3b. The 4-bit *x86_ilen* field returns the length of the x86 instruction. The 4-bit *uops_bytes* field returns the length of the generated uop(s) in the implementation ISA.

The *Flag_cmplx* bit is set if the x86-instruction being decoded is too complex for the hardware decoder. This escape mechanism keeps the hardware assist simple and fast by off-loading the complicated cases to software; for example, if the x86 instruction should happen to be more than 16 bytes (the size of the Fsrc register). The *Flag_cti* flag bit is set if the x86-instruction being processed is a control transfer instruction. After decoding, most x86-instructions are cracked into *uops* of no more than 16 bytes. Note that the 16-bit/32-bit fusible implementation ISA design implies that, only in a few rare cases, the 128b Fdst is too short to hold result *uops*; this is another case that is flagged as a complex instruction. Native *uops* in Fdst are written back to the code cache by a store operation. The rest of the loop does bookkeeping.

For architected state mapping, the CSR register can be mapped either to the same FP/media status register for x86 SIMD instructions, or to a separate implementation status register. Fsrc and Fdst are mapped to FP/media temporary registers F24 through F31.

For microarchitecture design, the new functional unit is located in the FP/media part of the processor core because it uses F registers to hold long x86 instructions and multiple *uops* (Figure 5.4). If implemented in superscalar style microarchitectures such as macro-op execution, the XLTx86 instruction would be dispatched to the FP/media instruction queue(s) and issued to the new functional unit via a FP/media issue port. XLTx86 can take multiple cycles to execute as do many other FP/media instructions. In our research, we assume XLTx86 takes four cycles. The x86 instruction bytes are supplied to the functional unit via streaming buffer and the generated uops are written back to memory directly without going through the data cache.

For circuit design, the functional unit for XLTx86 is essentially a simplified, one instruction wide, x86 decoder relocated in the execution stage of the FP/media core. For cost effectiveness, XLTx86 only needs to handle simple common cases. Frequent x86 instructions are handled by the decoding functional unit; complicated and rare x86 instructions set the *Flag_cmplx* flag in the CSR register to escape for VM software handling.

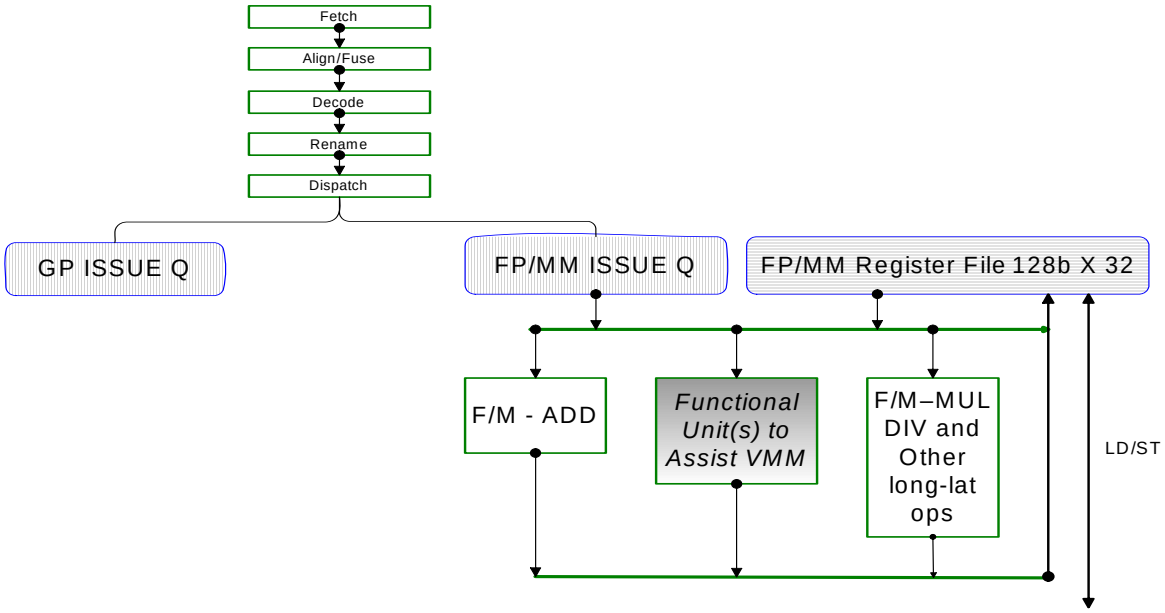


Figure 5.4 Hardware Accelerator microarchitecture design

The new instruction, XLTx86, speeds up BBT by accelerating the dominant part of the fetch, decode and crack, from tens of cycles for a software-only translator, to only a few cycles for BBT assisted by hardware. Meanwhile, because it is an instruction that provides a primitive operation (from the translator perspective), it offers the VMM flexibility and programmability beyond the dual mode decoders embedded in the pipeline front-end.

Compared with dual mode decoders, the functional unit only performs the BBT translation once for each basic block as long as the translation is not replaced. Although the BBT translation is still an extra overhead, the generated native code does not invoke further complex CISC decoding. Therefore, for non-hotspot emulation performance, the translations generated by BBT will likely perform similarly to the x86-mode enabled by the dual mode decoders. The BBT translation overhead, although significantly reduced by the XLTx86 assist, will likely to appear for cases where bursts of translations can occur.

The hardware complexity of the backend decoding functional unit is also less than the front-end dual mode decoders. As aforementioned, only frequent common cases are handled by the XLTx86 instruction, complex and rare cases are handled by software. Furthermore, just one such functional unit can achieve most of the translation performance boost. And a backend functional unit has very localized impact on the processor pipeline design.

The decoders for CISC instructions are power hungry circuits [54] and it is the complex first level decoding logic that consumes most of the power. In conventional x86 processors, these decoders need to turn on both levels and consume power whenever x86 instructions are fetched from memory hierarchy. In contract, dual mode decoders save energy by turning off the complex first-level decode stage when native code is executing during steady state. The decoder as the backend function unit, on the other hand, only consumes power when a new piece of code is executed for the first time. Hence, the backend function unit has similar energy implication as software-only VM systems.

5.3 Hardware Assists for Hotspot Profiling

In a staged translation system, early emulation stages also conduct online profiling to detect hotspots and trigger transitions to higher emulation stages. This profiling is performed by software in many VM systems. Although the major VM overhead is due to translation/emulation, the profiling can cause significant overhead once the DBT translation is assisted by hardware. In this section, we briefly discuss profiling in general. Then we emphasize simple hardware profilers/hotspot detectors that have already been proposed by related research efforts.

Program hotspot detection has long been performed by software profiling. The common software profiling mechanism is to instrument relevant instructions (or bookkeeping in an interpreter) to collect desired data. For program hotspot detection, control transfer instructions are instrumented so

that the execution frequency of basic blocks or paths can be tracked. Ball and Larus [14] proposed the first path profiling algorithm. This software instrumentation algorithm leads to 30-45% overhead for acyclic intra-procedure paths. When paths are extended to cross procedural or loop boundary, the overhead increases rapidly. TPP [72] and PPP [18] are proposed to attempt reducing the profiling overhead significantly without losing much accuracy and flexibility.

Hardware support for profiling first appeared as performance counters. Most recent AMD, IBM and Intel processors [51, 58, 120] are equipped with such simple assists to facilitate performance tuning on their server products.

Conte [31] introduced the profile buffer after the instruction retirement stage to monitor candidate branches. The proposed profile buffer is quite small, typically no more than 64 entries. Some compiler analysis and hint bit(s) are assumed to improve profiling accuracy. That is, to utilize this profiler, new binary needs to be generated by such a compiler.

Merten *et al* [96] proposed a larger (e.g. 4K-entry) Branch Behavior Buffer (BBB) and a hardware hotspot detector after the retirement stage. This enables their scheme to be transparent to applications and capable of profiling any legacy code. The hotspot threshold is a relative one – a branch needs to execute at least 16 times during the last 4K retired branches to qualify for a candidate branch. Detected hotspot(s) invoke software OS handlers for optimizations at runtime. Because this approach can be made transparent and cost-effective, we assume that an adapted version of this hardware hotspot detector is deployed in our VM system.

Vaswani *et al* [125] proposed a programmable hardware path profiler that is flexible and can associate microarchitectural events such as cache misses and branch misses with the paths being profiled. They ran 15-billion instructions for each benchmark to study more realistic workload behavior than many other research projects.

The HP Dynamo developed a simple but effective software hotspot detector. It counts branch target frequency while interpreting. Once a branch target exceeds the hot threshold, the interpreter forms a hot superblock using MRET (most recently executed tail) [13]. MRET captures frequent execution paths by probability, leading to a cheap and insightful profiler. However, the branch-target counter table maintained by the interpreter causes overhead and pollutes data cache.

5.4 Evaluation of Hardware Assists for Translation

The evaluation of hardware assists for translation was conducted with the *x86vm* simulation infrastructure. Because translation overhead affects mostly the VM startup performance, this evaluation focuses on how the assists improve VM startup behavior for Windows benchmarks.

To compare startup performance with conventional superscalar designs and to illustrate how VM system startup performance can be improved by the hardware assists, we simulate the following machine configurations. Detailed configuration settings are provided in Table 5.4.

- ***Ref*: *superscalar:*** A conventional x86 processor design serves as the baseline / reference. This is a generic superscalar processor model that approximates current x86 processors.
- ***VM.soft***: Traditional co-designed VM scheme, with a software-only two-stage dynamic translator (BBT and SBT). This is the state-of-the-art VM model.
- ***VM.be***: The co-designed x86 VM, equipped with pipeline backend functional units for the new XLTx86 instructions (Section 5.2).
- ***VM.fe***: The co-designed x86 VM, equipped with dual mode x86 decoders at the pipeline front-end to enable dual ISA execution. (Section 5.1).

Table 5.2 VM Startup Performance Simulation Configurations

	Ref: superscalar	VM.soft	VM.be	VM.fe
Cold x86 code	HW x86 decoders, no optimization	Simple software BBT, no opts	BBT assisted by the backend HW decoder.	HW Dual-mode decoders
Hotspot x86 code	HW x86 decoders, no optimization	Software hotspot optimizations	Perform the same hotspot optimization as in VM.soft, with HW assists.	
ROB, Issue buffer	36 issue queue slots, 128 ROB entries, 32 LD queue slots, 20 ST queue slots			
Physical Register File	128 entries, 8 Read ports, 5 Write ports	128 entries, 8 Read and 8 Write ports (2 Read and 2 Write ports are reserved for the 2 memory ports).		
Pipeline width	16B fetch width, 3-wide decode, rename, issue and retire.			
Cache Hierarchy	L1 I-cache: 64KB, 2-way, 64B lines, Latency: 2 cycles. L1 D-cache, 64KB, 8-way, 64B lines, Latency: 3 cycles. L2 cache: 2MB, 8-way, 64B lines, Latency: 12 cycles			
Memory Latency	Main memory latency: 168 CPU-cycles. 1 memory cycle is 8 CPU core cycles.			

The hot threshold in the VM systems is determined by Equation 2 (Chapter 3) and benchmark characteristics. For the *Windows* application traces benchmarked for this evaluation, all VM models *VM.soft*, *VM.be* and *VM.fe*, set the threshold at 8K. Note that the *VM.fe* and the *Ref: superscalar* schemes have a longer pipeline front-end due to the x86 decoders.

To stress startup performance and other transient phases for translation-based VM systems, we run short traces randomly collected from the ten Windows applications taken from the WinStone2004 Business benchmarks. For studies focused on accumulated values such as benchmark characteristics, we simulate 100-million x86 instructions. For studies that emphasize time variations, such as variation in IPC over time, we simulate 500-million x86 instructions and express time on a logarithmic scale. All simulations are set up for testing the *memory startup* scenario (Scenario 2 described in Chapter 3) to stress VM specific runtime overhead.

Startup Performance Evaluation of the VM Systems

Figure 5.5 illustrates the same startup performance comparison as Figure 3.1 in Chapter 3. Additionally, Figure 5.5 also shows startup performance for the VM models assisted by the proposed hardware accelerators. As before, the normalized IPC (harmonic mean) for the VM steady state is about 8% higher than the baseline superscalar in steady state.

The VM system equipped with dual mode decoders at the pipeline front-end (*VM.fe*) shows practically a zero startup overhead; performance follows virtually the same startup curve as the baseline superscalar because they have very similar pipelines for cold code execution. Once a hotspot is detected and optimized, the VM scheme starts to reap performance benefits. *VM.fe* reaches half the steady state performance gains (4%) in about 100-million cycles.

The VM scheme equipped with a backend functional unit decoder (*VM.be*) also demonstrates good startup performance. However, compared with the baseline superscalar, *VM.be* lags behind for the initial several millions of cycles. The breakeven point occurs at around 10-million cycles and the half performance gain point happens after 100-million cycles. After that, *VM.be* performs very similarly to the *VM.fe* scheme.

Figure 5.6 shows, for each individual benchmark, the number of cycles a particular translation scheme needs to reach the first breakeven point with the reference superscalar. The x-axis shows the benchmark names and the y-axis shows the number of million cycles a VM model needs to breakeven with the reference superscalar model. We label bars that are higher than 200-million cycles (to break even) with their actual values. Otherwise, a bar that is higher than 200-million cycles (but without a value label) means its VM model did not breakeven within the 500-million x86-instruction trace simulation.

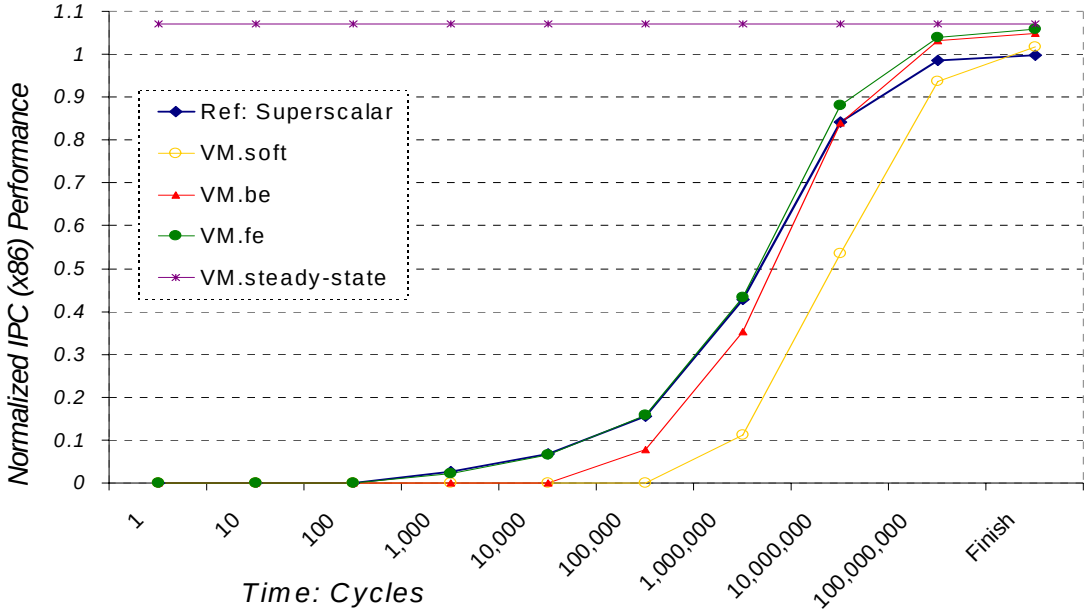


Figure 5.5 Startup performance: Co-Designed x86 VMs compared w/ Superscalar

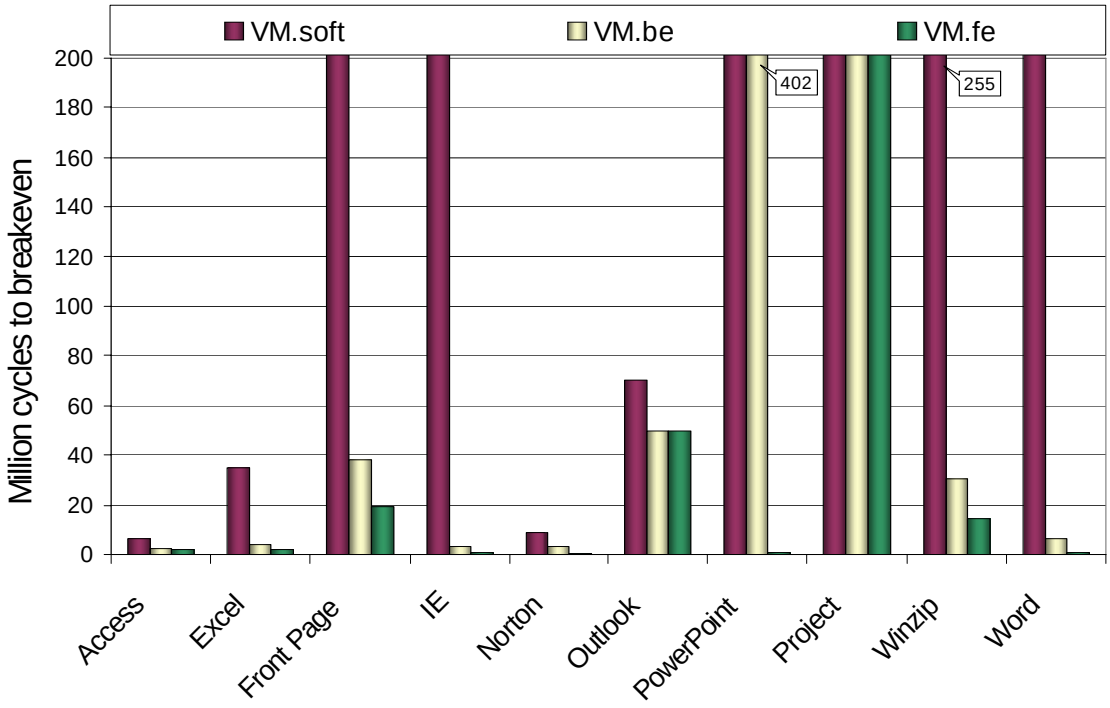


Figure 5.6 Breakeven points for individual benchmarks

It is clear from Figure 5.6 that, in most cases, using either the front-end or the backend assists can significantly reduce the VM startup overhead and enable VM schemes to break even with the reference superscalar within 50-million cycles. However, for the *Project* benchmark, the VM schemes cannot break even within the tested runs, though they do follow the performance of the reference superscalar closely (within 5%). Further investigation indicates that the VM steady state performance for *Project* is only 3% better than the superscalar. Thus the VM schemes take a longer time to collect enough hotspot performance gains to compensate for the performance loss due to initial emulation and translation.

It should be pointed out that, because of the different execution characteristics, VM systems may actually have multiple breakeven / crossover points for an individual benchmark's startup curve. This is not evident from the average curves in Figure 5.5. However, such transients occur even for systems using different memory hierarchies. As programs run longer, the superior VM steady state performance will start to take over, making a crossover point unlikely to repeat.

Performance Analysis of the Hardware Assists

It is straightforward to evaluate the startup performance improvement for *VM.fe* because its x86-mode execution is very similar to that of a baseline superscalar. On the other hand, the *VM.be* scheme translates cold code in a co-designed way and still involves VM software. Consequently, we investigate how VM software overhead is reduced after being assisted by the XLTx86 instruction. For background information, without hardware assist, the software-only baseline VM (*VM.soft*) spends on average 9.9% of its runtime performing BBT translation, during the first 100M dynamic x86 instructions,.

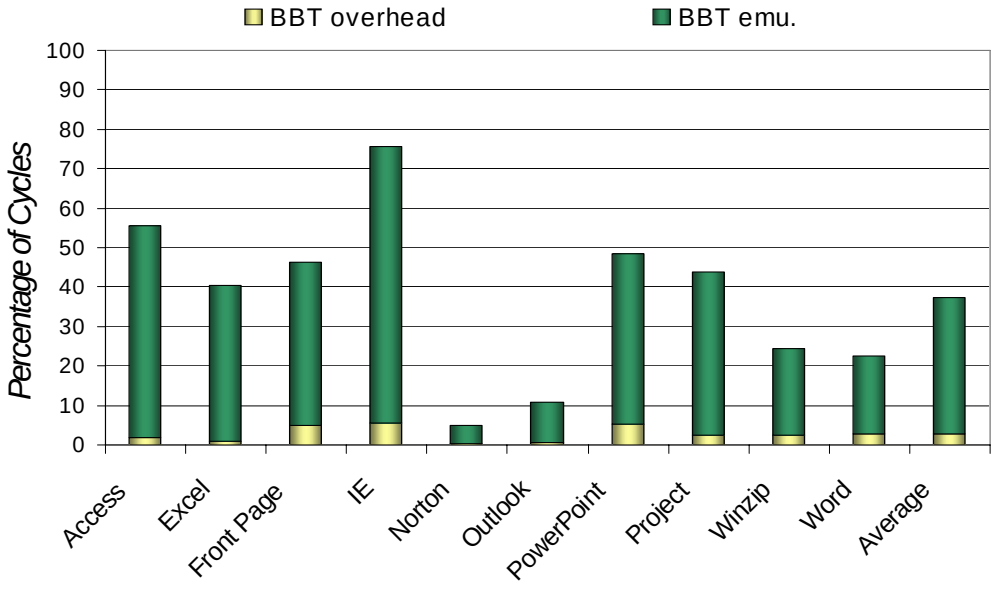


Figure 5.7 BBT translation overhead and emulation cycle time
(100M x86 instruction traces)

Figure 5.7 shows how VM cycles (for the *VM.be* scheme) are spent. For each benchmark, the lower bars (BBT overhead) represent the percentage of VM cycles spent for BBT translation and the upper bars (BBT emu.) indicate the percentage of cycles the *VM.be* executes basic block translations. The rest of the cycles are mostly spent for SBT translation and emulation with the optimized native hotspot code. To stress startup overhead, the data is collected for the first 100M x86 instructions for each benchmark.

It is evident from Figure 5.7 that after being assisted by the new XLTx86 instruction at the pipeline backend, the average BBT translation overhead is reduced to about 2.7%; about 5% at worst. Further measurements indicate that the software-only BBT spends 83 cycles to translate each x86-instruction (including all BBT overhead, such as chaining and searching translation lookup table). In contrast, *VM.be* needs only 20 cycles to do the same operations.

After BBT translation, the *VM.be* scheme spends 35% of its total cycles (*BBT.emu* bars in Fig. 10) executing BBT translations. The execution of BBT translations is less efficient than that of SBT translations. However, this BBT emulation does not lose much performance because the BBT translations run fairly efficiently (On average 82~85% IPC performance of SBT optimized code). This IPC performance is only slightly less than the baseline superscalar design. And for program startup transients, cache misses tend to dilute CPU IPC performance differences.

The rest of the *VM.be* cycles (*VM.fe* is similar) is spent in SBT translation (3.2%) and native execution with the SBT translations (59%). The optimized SBT translations improve overall performance by covering 63% of the 100-million dynamic x86 instructions. For 500-million x86 instruction runs, the hotspot coverage rises to 75+% on average and is projected to be higher for full benchmark runs.

Energy Analysis of the Hardware Assists

A software-based co-designed VM does not require complex x86 decoders in the pipeline as in conventional x86 processors. This can provide significant energy savings (one of the motivations for the Transmeta designs [54, 82]). However, when hardware x86 decoder(s) are added as assists, they consume energy. Fortunately, this energy consumption can be mitigated by powering off the hardware assists when they are not in use.

To estimate the energy consumption, we measure the activity of the hardware x86 decoding logic. The activity is defined as the percentage of cycles the decoding logic needs to be turned on. Figure 5.8 shows the x86 decoder activity for the four machine configurations. The x-axis shows the cycle time on logarithmic scale and the y-axis shows the *aggregate* decoding logic activity. This figure assumes that the decoders need to be turned on initially, as the system runs, the decoders can be turned on and off quickly based on usage.

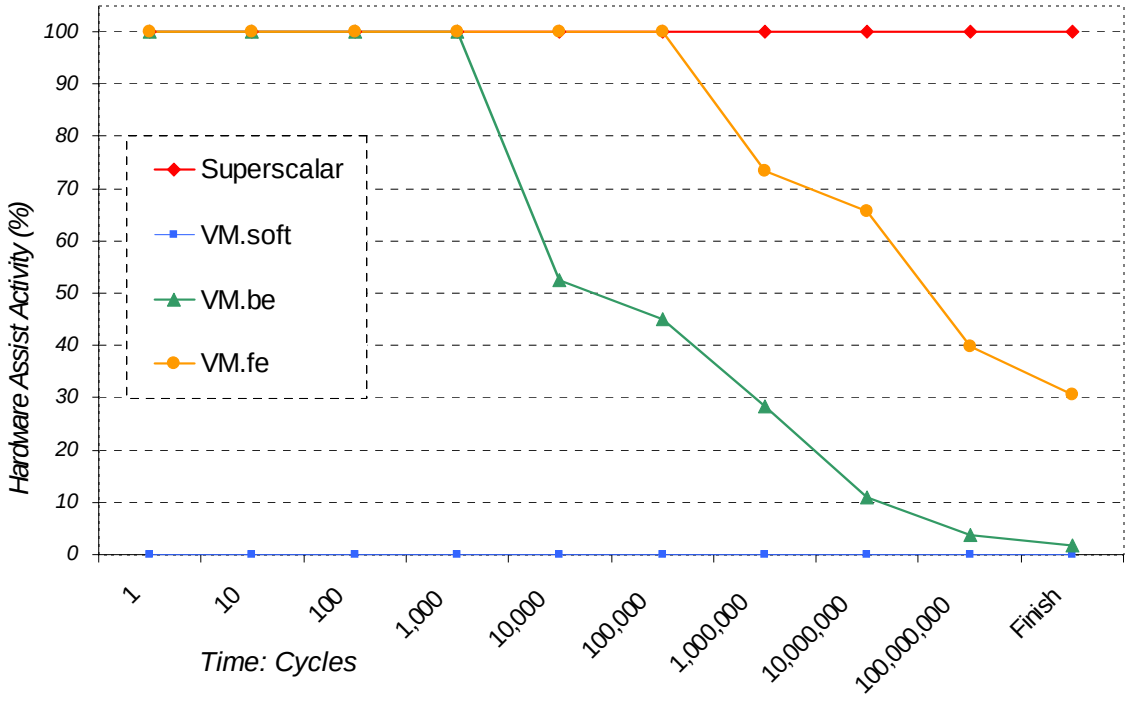


Figure 5.8 Activity of hardware assists over the simulation time

For most conventional x86 processors, x86 decoders are always on (Pentium 4 [58] is the only exception). In contrast, for the *VM.be* scheme, the hardware assist activity quickly decreases after the first 10,000 cycles. It becomes negligible after 100-million cycles. Considering that only one decoder is needed to implement XLTx86 in the *VM.be* scheme, energy consumption due to x86 decoding is significantly mitigated. For the *VM.fe* model, the dual mode decoders at the pipeline front-end need to be active whenever the VM is not executing optimized hotspot code (and the VMM code, to be more exact). The decoders' activity also decreases quickly, but much later than a *VM.be* scheme as illustrated in the figure. Because the *VM.fe* scheme executes non-hotspot code rather than translating once (as in *VM.be*), it will be more demanding on the responsiveness of turning on and off the dual mode decoders.

5.5 Related Work on Hardware Assists for DBT

In Section 5.4 we discussed on-the-fly profiling, which is an important part of VM DBT systems both for identifying hotspot code and for assisting with certain optimizations. This section discusses related work on hardware assists for binary translation/optimization.

The Transmeta Efficeon designers implemented an *execute* instruction that allows native VLIW instructions to be constructed and executed on-the-fly [83]. This capability was added to improve the performance of the CMS interpreter. At the pipeline backend, there are two execution units added. However, details about the new execution unit design are not published. In contrast, we propose special hardware assists to accelerate the BBT translation and then save the translated code in a code cache for reuse.

The fill unit [50, 95] is one of the early hardware proposals for runtime translation or optimization. A fill unit is a non-architected transparent module that constructs a translation unit (typically, trace) and then performs limited optimization. The Intel Pentium 4 processor trace cache [58] is an implementation of a similar hardware scheme.

The Instruction Path Coprocessor [25] is a programmable coprocessor that optimizes a core processor's instructions to improve execution efficiency. The coprocessor is demonstrated to perform several common dynamic optimizations such as trace formation, trace scheduling, register-move optimization and prefetching. A later proposal, PipeRench implementation of the instruction path coprocessor [26], reduces the complexity of the coprocessor design.

Dynamic strands [110] employ a hardware engine to form ILDP strands [77] for a strand aware engine. CCG [27] and many other research proposals [2, 104, 114] assume a hardware module to perform runtime code optimization.

The rePLay [104] and PARROT [2] projects employ a hardware hotspot detector to find program hotspots. Once a hotspot is detected, it is optimized via hardware and stored in a small on-chip frame/trace cache for optimized hotspot execution. The hardware optimizer is a coprocessor-like module located after the retirement stage of the main processor pipeline. As hotspot instructions are detected and collected in a hotspot buffer, the optimizer can execute a special program, perhaps stored in an on-chip ROM similar to the micro-code stores [58, 74]. This special program is written in the optimizer's small instruction set that consists of highly specialized operations such as pattern matching and dependence edge tracking etc.

In our research, we explore hardware assists that are integrated into the co-designed processor pipeline. This requires simpler hardware than a full-blown coprocessor or hardware optimizer. Yet, the software translator owns the full programmability of the main processor's ISA. The translator runtime overhead is mitigated via the combination of strategies discussed in Chapter 3 and simple hardware assists discussed in this chapter. Furthermore, in our VM system, translated code is held in a large main memory code cache.

Chapter 6

Putting It All Together: A Co-Designed x86 VM

In the previous three chapters, we explored and achieved promising results for some major components of a co-designed VM system, namely, a DBT translation strategy, translation software algorithms, and translation hardware assists. However, individual component designs do not necessarily combine together to produce an overall efficient system design. To demonstrate a processor design paradigm, the most convincing evidence is a complete, integrated processor design that provides superior performance and efficiency.

In this chapter, I explore system level design through synergetic integration of the technologies explored in the previous chapters. The detailed example design must tackle many thorny challenges for contemporary processors. Section 6.1 discusses high level trade-offs for the co-designed x86 processor architecture. Section 6.2 details the microarchitecture of the macro-op execution engine. Section 6.3 evaluates the integrated, hardware/software co-designed x86 processor within the limit of our *x86vm* framework. Section 6.4 compares this design with the related real world processor designs and research proposals.

6.1 Processor Architecture

There are a few especially difficult issues that pose challenges to future processor designs. First, processor efficiency (performance per transistor) is decreasing. Although performance is improving, but the improvements are not proportional to the amount of hardware resources and complexity invested. Additionally, the complexity of the design process itself is increasing. Time-to-market tends to be longer today than a decade ago.

Second, power consumption has become critical. Power consumption not only affects energy efficiency, but also affects many other cost aspects of system designs, for example, the extra design complexity for the power distribution system, cooling, and packaging. High power consumption also leads to thermal issues that affect system reliability and lifetime.

Third, legacy features are making processor designs less efficient. As the x86 instruction set has become the *de facto* standard ISA for binary software distribution, efficient x86 design is becoming more important. Moreover, as with most long term standard interfaces, the x86 contains legacy features that complicate hardware design if implemented directly in hardware. Many of these legacy features are rarely used in modern software distributions.

The primary objective for the example co-designed x86 processor presented here is an efficient design that brings higher performance at lower complexity. The overall design should tackle the design challenges listed above. Furthermore, for practical reasons, it is especially valuable for a design team to consider a design that carries minimum cost and risk for an initial attempt at a new implementation paradigm.

Clearly, an enhanced and more efficient dynamic superscalar pipeline design is an attractive direction to explore. On one hand, dynamic superscalar is the best performing and the most widely microarchitecture for general purpose computing. On the other hand, dynamic superscalar processors

are challenged by bottlenecks in several pipeline stages: for example, the fetch mechanism, the issue logic, and the execution/result forwarding datapath.

I propose and explore the *macro-op execution* microarchitecture (detailed in Section 6.2) as an efficient execution engine. It simplifies and streamlines the critical pipeline stages given in the preceding paragraph. It is not only an enhanced, but also a simplified dynamic superscalar micro-architecture that implements the fusible ISA for efficiency. Because it is a simplified superscalar pipeline, the hardware design is quite similar to current processor designs. Then, the remaining major processor design trade-off centers on dynamic binary translation which performs the mapping from the x86 ISA to the fusible ISA.

Figure 6.1 illustrates the trade-off between overall system complexity and DBT runtime over-head. A state-of-the-art VM design should use the software solution for DBT to emphasize simplicity for CPU intensive workloads. A conventional high performance x86 processor design might select the hardware solution to avoid bad-case scenarios mentioned in Chapter 1.

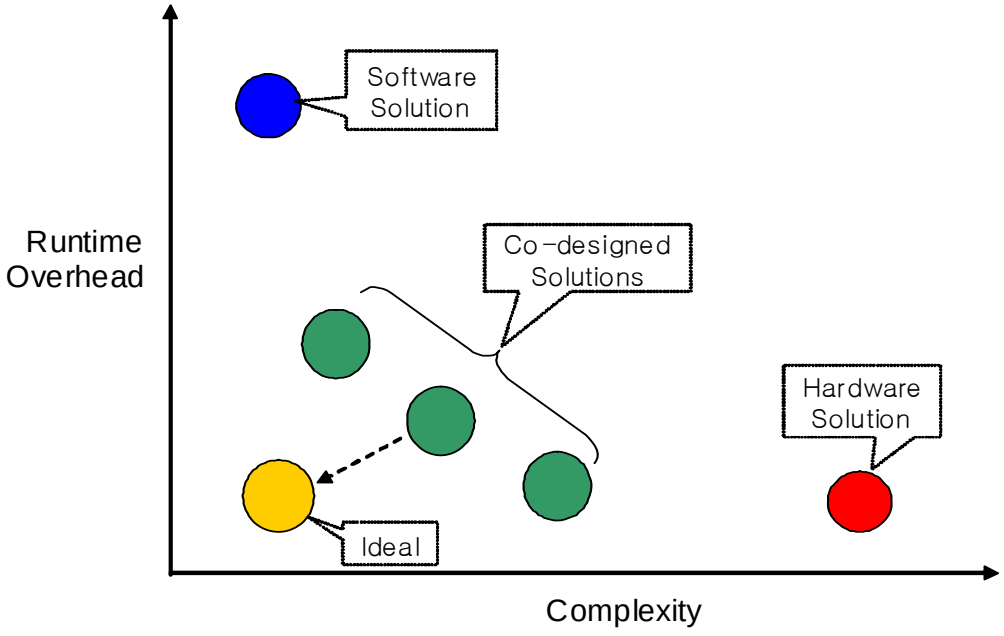


Figure 6.1 HW/SW Co-designed DBT Complexity/Overhead Trade-off

The preceding chapters have developed the key elements for a hardware/software co-designed DBT system. In fact, there are multiple possible hardware assisted DBT systems and this is reflected in Figure 6.1. For example, Chapter 5 proposed and evaluated two design points, dual mode decoders at the pipeline front-end and special functional unit(s) at the pipeline backend. Either of these can provide competitive startup performance as illustrated in Chapter 5.

For the example x86 processor design, we choose the dual mode decoders because it has very competitive startup performance to conventional x86 processor designs. Furthermore, it provides a more smooth transition from conventional designs to the VM paradigm. The downside is that this design point does not remove as much hardware complexity as the backend functional unit(s) solution. However, it does provide the same hotspot optimization capability as other VM design points. And these runtime optimizations would be very expensive if implemented in hardware.

Because the dual mode decoders have two modes for x86 and fusible ISA instructions, the macro-op execution pipeline is conceptually different for x86 mode and macro-op mode (Figure 6.2). The difference is due to the extra x86 vertical decode stages which first identify individual x86 instructions, decode, and crack them into fusible ISA micro-ops. The VMM runtime controls the switch between these two modes. Initially, all x86 software runs through the x86 mode pipeline. Once hotspots are detected, the VMM transfers control to the optimized macro-op code for efficiency.

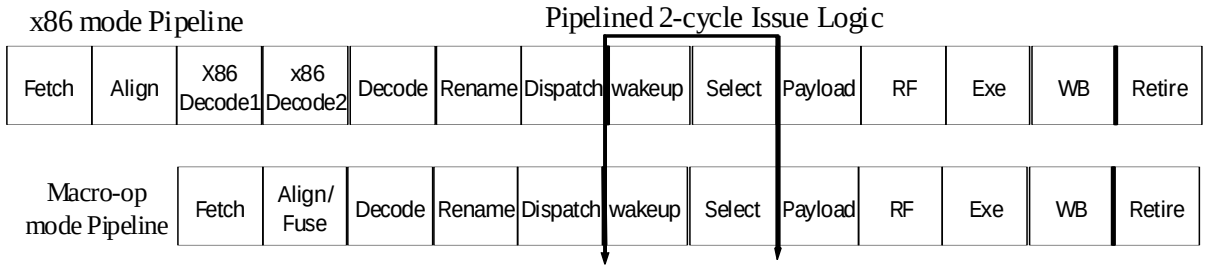


Figure 6.2 Macro-op execution pipeline modes: x86-mode and macro-op mode

The x86 mode pipeline is very similar to current high performance x86 processors, except that the instruction scheduler is also pipelined for higher clock speed and reduced scheduler complexity. Compared with the macro-op mode pipeline, the additional x86 mode pipeline stages consume extra power for the x86 “vertical” decode and charge more penalty cycles for branch mispredictions.

The macro-op mode pipeline is an enhanced dynamic superscalar processor for performance and efficiency. The VMM runtime software and the code cache which holds translated and optimized macro-op code for hotspots occupy and conceal a small amount of physical memory from the system. However, because the dual mode co-designed processor does not need to handle startup translations, there is no need for a BBT code cache, which is much larger than the hotspot code cache. The details of the macro-op execution pipeline are expanded upon in the next section.

6.2 Microarchitecture Details

The macro-op execution pipeline executes fused macro-ops throughout the entire macro-op mode pipeline. There are three key issues, macro-op fusing algorithms, macro-op formation, and macro-op execution. The co-designed VM software conducts macro-op fusing (Chapter 4). The co-designed hardware pipeline performs macro-op formation and macro-op execution at the pipeline front-end and backend respectively.

6.2.1 Pipeline Frond-End: Macro-Op Formation

The front-end of the pipeline (Figure 6.3) 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 fusible bits marked by the dynamic binary translator. After the formation of macro-ops, the pipeline front-end also allocates bookkeeping resources such as ROB, LD/ST queue and issue queue slots.

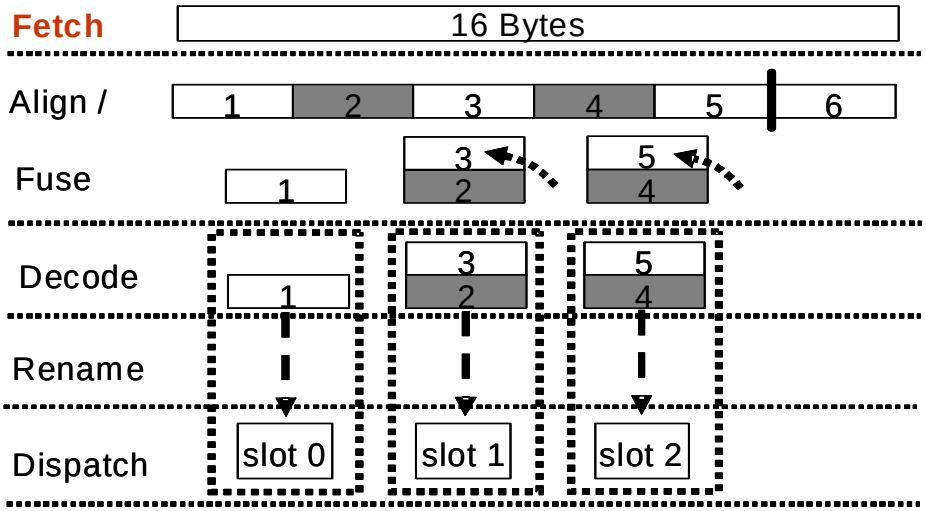


Figure 6.3 The front-end of the macro-op execution pipeline

Fetch, Align and Fuse

Each cycle, the fetch stage brings in a 16-byte chunk of instruction bytes from L1 instruction cache. The effective fetch bandwidth, four to eight micro-ops per cycle, is a good match with the effectively wider pipeline backend. After fetch, an align operation recognizes instruction boundaries. In x86 mode, x86 instructions are routed directly to the first level of the dual-mode decoders. In macro-op mode, the handling of optimized native 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. Micro-ops bypass the first level of the decoders and go to the second level directly. The first bit of each micro-op, the fusible bit, indicates whether 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.

While fetching, branch predictors help determine the next fetch address. The proposed branch predictor is configured similarly to the one used in the AMD K8 Opteron processor [74]. Specifically, it combines a 16K-entry bimodal local predictor with a 16K-entry global history table via a 16K

combining table. The global branch history is recorded by a 12-bit shift register. BTB tables are larger and more expensive, especially for 64-bit x86 implementations. The BTB (4-way) has 4K entries and the RAS (return address stack) has 16 entries, larger than the AMD K8's 2K BTB and 12-entry RAS. Both x86 branches and native branches are handled by this predictor.

Instruction Decode

In x86 mode, x86 instructions pass through both decode levels and take three or more cycles (similar to conventional x86 processors [37, 51, 74]) for instruction decoding and cracking. For each pipeline lane, the dual mode decoder has two simple level-two micro-op decoders that can process up to two micro-ops cracked from an x86 instruction. As with most current x86 processors, complex x86 instructions that crack into more than two micro-ops may need to be decoded alone for that cycle, and string instructions need a microcode table for decoding.

In macro-op mode, RISC-style micro-ops pass through the second level decoding stage only and take one cycle to decode. For each pipeline lane, there are two simple level-two micro-op decoders that handle pairs of micro-ops (a fused macro-op). These micro-op decoders decode the head and tail of a macro-op 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 performance advantage of a shorter pipeline for macro-ops can be significant for workloads that have a significant number of branch mispredictions.

After the decode stage, the micro-ops for both x86 mode and macro-ops look similar to the pipeline except that none of x86 mode micro-ops are fused. Here, an interesting optimization, similar to one used in the Intel Pentium M decoders [51], is to use the level-one decoder stage(s) to fuse simple, consecutive micro-ops when in x86 mode. However, this will add extra complexity and only benefit code infrequently. This possibility is not implemented in our design here.

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. However, macro-ops simplify the rename process (especially source operand renaming) because (1) the known dependence between a macro-op head and tail does not require intra-group dependence checking or a map table access, and (2) the total number of source operands per macro-op is two, which is the same for a single micro-op in a conventional pipeline.

Macro-op dependence translation converts register names into *macro-op name*s 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. Moreover, 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 in program order into available issue queue(s) and ROB entries at the dispatch stage. Memory accesses are also inserted into LD/ST queue(s). Because the two micro-ops in a fused pair have at most two source operands and occupy a single issue queue 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 book-keeping information are stored in the payload RAM [21].

6.2.2 Pipeline Back-End: Macro-Op Execution

The back-end of the macro-op execution pipeline performs out-of-order dataflow execution by scheduling and executing macro-ops as soon as their source values become available. This kernel part of a dynamic superscalar engine integrates several unique execution features.

Instruction (Macro--op) Scheduler

The macro-op scheduler (issue logic) is pipelined [81] 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 by eliminating the sequencing point at the payload RAM stage as proposed in [81]. Thus, it eliminates the necessity of blocking the select logic for macro-op tail micro-ops.

Operand fetch: Payload RAM Access and Register File

After issue, a macro-op accesses the payload RAM to acquire the physical register identifiers, opcode(s) and other necessary information needed for execution. Each payload RAM line has two entries for the two micro-ops fused into a macro-op. Although this configuration will increase 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 read (the payload stage) and write (the dispatch stage) accesses, increasing the effective bandwidth. For example, a 3-wide dispatch / execution 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 to fetch the source operand values for two fused operations. Because the maximum number of 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 the head and tail, and increasing the probability that both register identifiers of a macro-op are actually used. Furthermore, because we decided to employ collapsed 3-1 ALU units at the execution stage (described in the next subsection), 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.

Our macro-op mode does not improve register write port utilization, and requires the same number of 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 binary translation time. We leave this to future work. In fact, as the fusing profile indicates, only 6% of all instruction entities in the macro-op execution pipeline actually need to write two destination registers (Section 4.7 and [63]).

Execution and Bypass Network

Figure 6.4 illustrates the data paths in 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 (if a tail exists) micro-ops are routed to a collapsed 3-1 ALU [71, 91, 106] 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 2-1 normal ALU [92, 106].

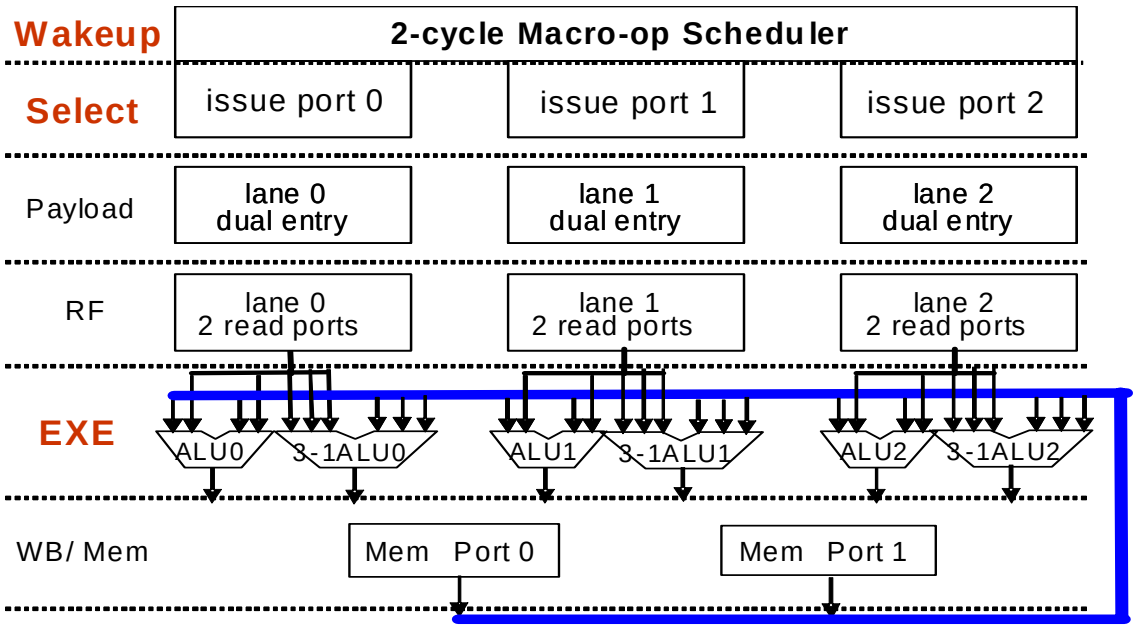


Figure 6.4 Datapath for Macro-op Execution (3-wide)

For a conventional superscalar execution engine with *n* ALUs, the ALU-to-ALU bypass network needs to connect all *n* ALU outputs to all *2*n* inputs of the ALUs. Each bypass path needs to drive at least *2*n* loads. Typically there are also bypass paths from other functional units such as memory ports. The implication is two-fold. (1) The consequent multiple input sources (more than *n+3*) 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 the matter worse, as operands are extended to 64-bit, the ALU areas and wires also increase significantly. In fact, wire issues and pressures on register file led the DEC Alpha EV6 [75] to adopt a clustered microarchitecture design and the literature verifies that the bypass latency takes a significant fraction of ALU execution cycle [48, 102]. There is a substantial body of related work (e.g. [46, 77, 102]) that addresses such wiring issues.

The novel combination of a 2-cycle pipelined scheduler and 3-1 collapsed ALUs enables the removal of the expensive ALU-to-ALU operand bypass network without IPC performance penalty. Because all the head and tail ALU operations are finished in one cycle; there is no need to forward the newly generated result operands to the inputs of the ALU(s) since tail operations finished one cycle sooner than the dataflow graph and dependent operations are not yet scheduled by the pipelined issue logic. There is essentially an "extra" cycle for writing results back to the register file. The removal of the operand forwarding/bypass network among single-cycle ALUs reduces pipeline complexity and power consumption.

Functional units that have multiple cycle latencies, e.g. cache ports, still need a bypass network as highlighted in Figure 6.4. However, the complexity of the bypass paths for macro-op execution is much simpler than a conventional processor. In macro-op execution, the bypass network only connects outputs (from multi-cycle functional units) to inputs of the ALU(s). In contrast, a conventional superscalar design having a full bypass network needs to connect across all input and output ports for all functional units.

Figure 6.5 represents 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 that fuse conditional branches with their condition test operations will resolve the branches 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. Early address resolution helps memory disambiguation, resulting in fewer expensive replays due to detected memory consistency violations in multiprocessors.

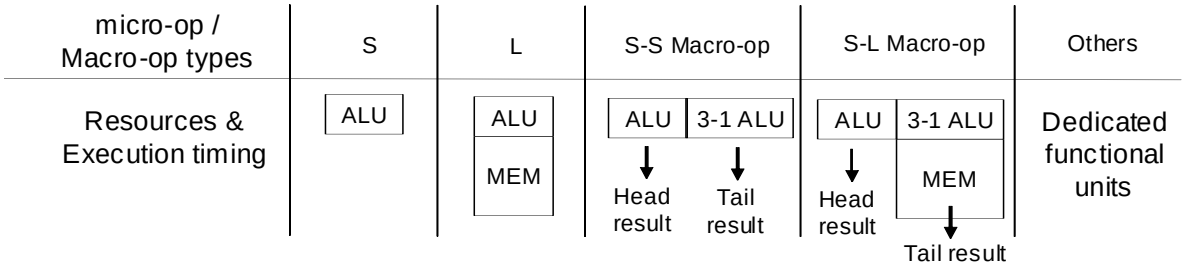


Figure 6.5 Resource requirements and execution timing

Writeback

After execution, values generated by ALU operations are written back to the register file via reserved register write ports. Memory accesses are different; a LD operation may miss in the cache hierarchy and a ST operation may only commit values to memory after the ST has retired. As with most modern processors [37, 51, 58, 74], the macro-op execution pipeline has two memory ports. Therefore, two register write ports are reserved for LD operations and two register read ports are reserved for ST operations.

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 does not produce a value, thus it is not fused as a head in a macro-op. 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) [85]. Therefore, the VM runtime software can also enable aggressive optimization(s) without losing intrinsic binary compatibility.

6.3 Evaluation of the Co-Designed x86 processor

Pipeline models

To analyze and compare the co-designed processor with conventional x86 superscalar designs, I simulated two primary microarchitecture models. The first, *baseline*, models a conventional dynamic superscalar design with single-cycle issue logic. The second model, labeled *macro-op*, is the proposed co-designed x86 microarchitecture. Simulation results were also collected for a version of the baseline model with pipelined two-cycle issue logic, which can be used to estimate the x86 mode operation (startup behavior) of the dual mode co-designed x86 processor.

The baseline model attempts to capture the performance characteristics similar to a Pentium-M or AMD K7/K8 implementation. However, we were only able to simulate an approximation of these best performing x86 processors. First, the baseline model uses fusible ISA micro-ops instead of the proprietary Intel or AMD micro-ops (which we do not have access to for obvious reasons). Also it does not fuse micro-ops the way Pentium M does, strictly speaking, but rather has significantly wider front-end resources to provide a performance effect similar to Pentium-M micro-op fusion or AMD Macro-Operation. In the baseline model, 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 *1.5*n* wide pipeline front-end. For example, the four-wide baseline can crack four x86 instructions into up to six micro-ops, which are then passed through the front-end pipeline. The micro-ops in the baseline model are scheduled and issued separately as in the current AMD or Intel x86 processors.

Microarchitectural resources for the three microarchitectures are listed in Table 6.1. Note that we reserve two register read ports for stores and reserve two write ports 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.

Table 6.1 Microarchitecture Configurations

	BASELINE	*BASELINE PIPELINED*	MACRO-OP
ROB Size	128	128	128
Retire width	3,4	3,4	2,3,4 macro-ops
Scheduler Pipeline Stages	1	2	2
Fuse RISC-ops?	No	No	Yes
Issue Width	3,4	3,4	2,3,4 macro-ops
Issue Window Size	Variable. Sample points: from 16, up to 64. Effectively larger for the macro-op mode.		
Functional Units	4,6,8 integer ALU, 2 MEM R/W ports, 2 FP ALU		
Register File	128 entries, 8,10 Read ports, 5,6 Write ports		128 entries, 6,8,10 Read 6,8,10 Write ports
Fetch width	16-Bytes x86 instructions		16B fusible micro-ops
Cache Hierarchy	L1 I-cache: 4-way 32KB, 64B cache lines latency: 2 cycles. L1 D-cache: 4-way 32KB, 64B lines, latency: 2 cycles + 1 cycle AGU. L2 cache (unified): 8-way 1 MB, 64B cache lines, latency: 8 cycles.		
Memory Latency	200 CPU cycles One memory cycle is 8 CPU clock cycles		

Performance

SPEC2000 is a standard benchmark for evaluating CPU performance, Figure 6.6 first shows the relative IPC performance for SPEC2000 integer benchmarks. The x-axis shows issue window sizes ranging from 16 to 64. The y-axis shows IPC performance that is normalized with respect to a 4-wide baseline x86 processor with a size 32 issue window (These normalized IPC values are close to the absolute values; the harmonic mean of absolute x86 IPC is 0.95 for the 4-wide baseline with issue window size 32). Five bars are presented for configurations of 2, 3, and 4-wide macro-op execution model; 3 and 4-wide baseline superscalar.

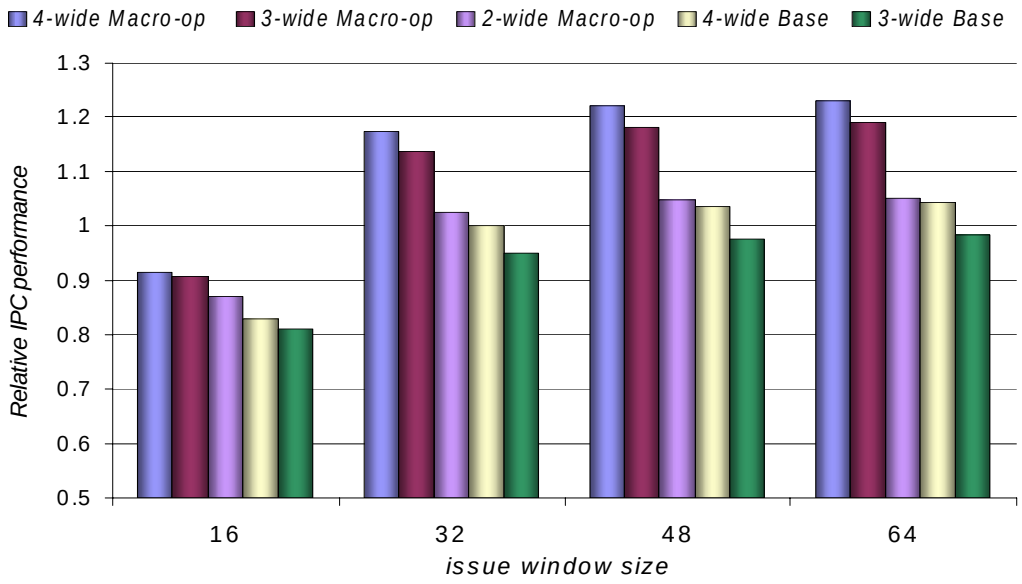


Figure 6.6 IPC performance comparison (SPEC2000 integer)

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. 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 pipelines the issue stage by processing macro-ops. Hence, we can argue that the macro-op model should be able to support either a significantly higher clock frequency or a larger issue window for a fixed frequency, thus giving the same or better IPC performance as a conventional four-wide processor. It assumes no deeper a pipeline than the baseline model, and in fact it reduces pipeline depth for steady state by removing the complex first-level x86 decoding/cracking stages from the critical branch misprediction redirect path. On the other hand, if we pipeline the issue logic in the baseline design for a faster clock, there is an average IPC performance loss at about 5%.

If we consider the performance data in terms of IPC alone, the four-wide co-designed x86 processor performs nearly 20% better than the baseline four-wide superscalar primarily due to its runtime binary optimization and its efficient macro-op execution engine which has an effectively larger window and issue width. As illustrated in Section 4.7, macro-op fusing increases operation granularity by 1.4 for SPEC2000 integer benchmarks. 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.

On the other hand, such superior CPU performance will be harder to achieve for whole system workloads such as the WinStone2004 because system workloads also stress other system resources, for example, memory system, I/O devices and OS kernel services. The improved CPU performance will be diluted.

Figure 6.7 plots the same IPC performance for whole system Windows application traces collected from WinStone2004. The performance test runs are 500M x86 instructions. Clearly, the co-designed VM system performance improvement is less than for SPEC2000, though still significant. And there are two further observations for the Windows workloads.

The first is that the whole VM system performance is diluted by the startup phase. VM IPC performance is improved by about 5% (4-wide), rather than the 8% IPC improvement for hotspot code alone. In contrast, the hotspot IPC performance figure for SPEC2000 is essentially the same as the whole VM system IPC performance. As pointed out previously, the Windows workloads tend to have a larger code footprint that will cause more VM startup overhead and collect less hotspot benefits than SPEC2000 benchmarks.

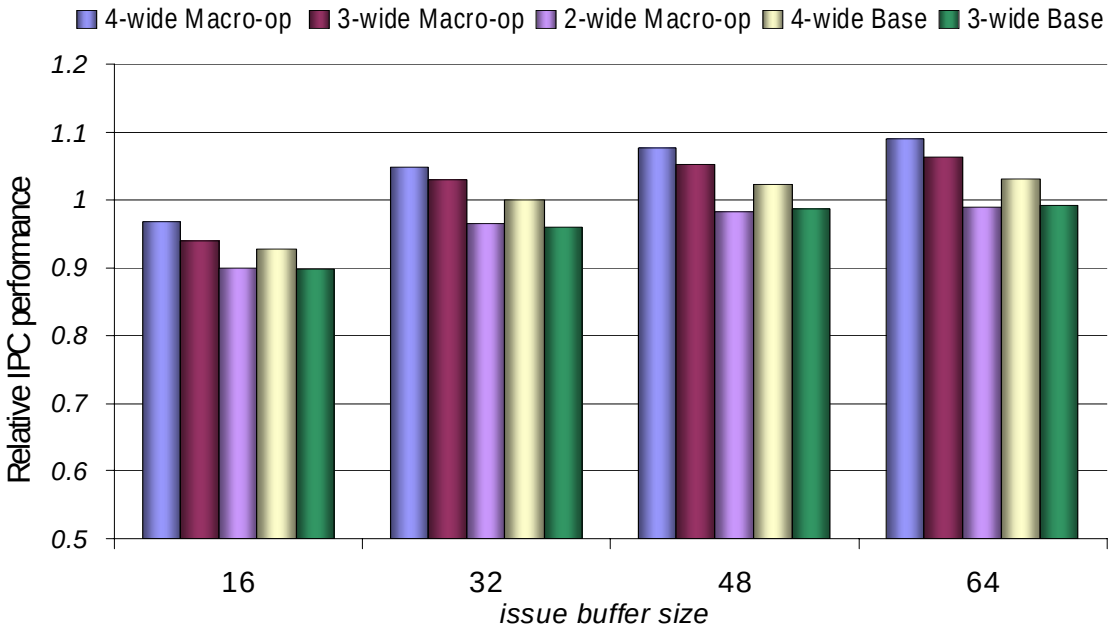


Figure 6.7 IPC performance comparison (WinStone2004)

The second observation is that the IPC performance improvement for a two-wide macro-op execution engine can match a three-wide baseline, but not a four-wide baseline as for SPEC2000. This is mainly caused by fewer fused macro-ops for the WinStone2004 Business Suite. Otherwise, the IPC performance *trends* look very similar to that for SPEC2000.

The major performance-boosting feature in the co-designed x86 processor is macro-op fusing which is performed by the dynamic translator. The macro-op fusing data presented in Section 4.7 illustrated that on average, 56% of all dynamic micro-ops are fused into macro-ops for SPEC2000, and 48% for Windows applications. Most of the non-fused operations are loads, stores, branches, floating point and NOPs. Non-fused single-cycle integer ALU micro-ops are only 6~8% of the total, thus greatly reducing the penalty due to pipelining the issue logic.

Performance experiments were also conducted with the single-pass fusing algorithm in [62]. And it actually shows average IPC slowdowns for SPEC2000 when compared with a baseline at the same width. The greedy fusing algorithm there does not prioritize critical dependences and single-cycle ALU operations. For Windows applications, single-pass fusing can approach the IPC performance of a same-width baseline.

Performance Analysis

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

- Fusing of dependent operations allows a larger effective window size and issue width, which is one of our primary objectives.
- 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 replicated code, however, due to the code duplication that occurs as superblocks are formed.
- Fused operations lead naturally to collapsed ALUs having a single cycle latency for dependent instruction pairs. Due to pipelined (two cycle) instruction issue queue(s), the primary benefit is simplified result forwarding logic, not IPC performance. However, there are some performance advantages because the latency for resolving conditional branch outcomes and the latency of address calculation for load/store instructions are sometimes reduced by a cycle.
- Because the macro-op mode pipeline only has to deal with RISC-style micro-ops, the pipeline 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.

0. Baseline: as before

1. ***M0***: Baseline plus superblock formation and code caching (but no translation).

2. ***M1***: M0 plus fused macro-ops; the pipeline length is unchanged.

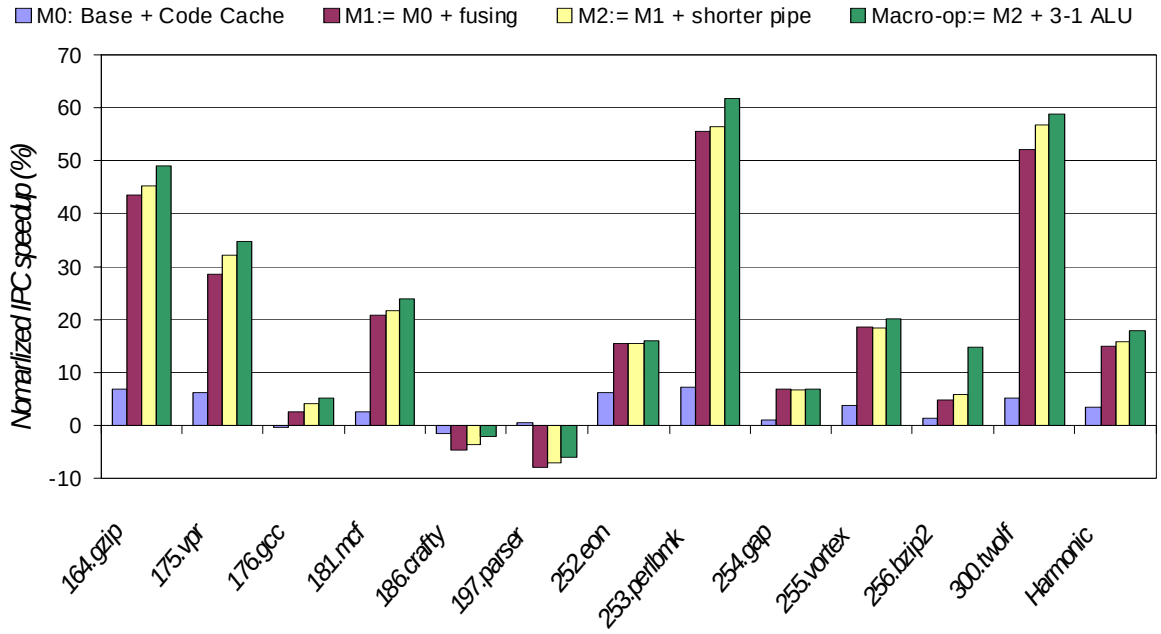
3. ***M2***: M1 with a shortened front-end pipeline to reflect the simplified decoders for macro-op mode.

4. ***Macro-op***: as before – M2 plus collapsed 3-1 ALU.

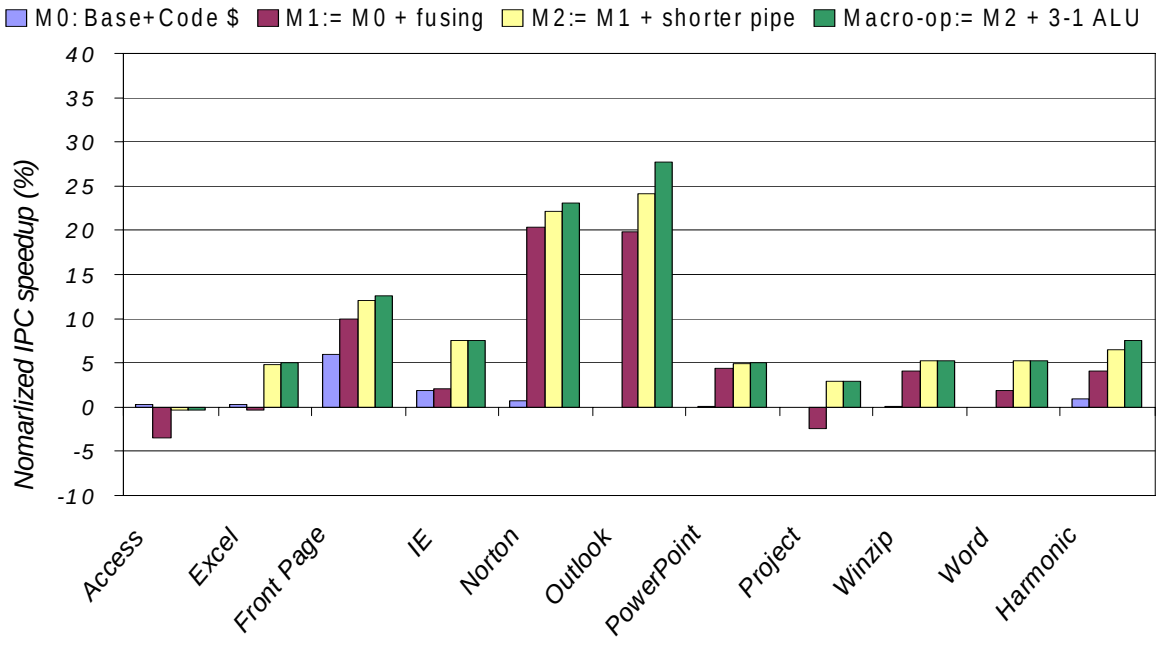
All of these configurations were simulated for the four-wide co-designed x86 processor configuration featuring the macro-op execution engine, and results are normalized with respect to the four-wide baseline (Figure 6.8).

The M0 configuration shows how a hotspot code cache helps improve performance via code re-layout. The average improvement is nearly 4% for SPEC2000 and 1% for WinStone2004. 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.

It is important to note that in the M0, the code has not been translated. There are two types of code expansions that can take place (1) code expansion due to superblock tail duplication, (2) code expansion due to translation. Hence, only the first type of expansion is reflected in the M0 design. The reason for using the M0 design point (superblock formation for a conventional superscalar) was so that the fused design does not get "credit" for code straightening when the other optimizations are applied. Performance effects of code expansion due to translation are counted in the M1 bar.



(a) SPEC2000 integer



(b) WinStone2004 Business Suites

Figure 6.8 Contributing factors for IPC improvement

The performance of M1 (when compared with M0) illustrates the gain due mainly to macro-op fusion. This is the major contributor to IPC improvements and is more than 10% on average for SPEC2000. For WinStone2004, the fusing compensates for most of the negative performance effects due to the pipelined scheduler and improves over baseline superscalar by 3%.

With regard to translation expansion, the second type of expansion noted above, the translated code is 30~40% bigger than the x86 code. However, in SPEC2000 integer, only *gcc*, *crafty*, *eon* and *vortex* are sensitive to code expansion with the 32K L1 I-cache. Other benchmarks show close to zero I-cache miss rates for baselines and VM models. For *gcc*, *crafty*, *eon* and *vortex*, VM models have higher I-cache miss rates. This observation helps to explain the IPC loss for crafty, for example.

The average performance gain due to a shortened decode pipeline is nearly 1% for SPEC2000. However, this gain will be higher for applications where branches are less predictable. For example, for WinStone2004, it is about 2%.

Finally, the performance benefit due to a collapsed ALU is about 2.5% for SPEC2000 and 1% for WinStone2004. As noted earlier these gains are from reduced latencies for some branches and loads because the ALU result feeding these operations is available a cycle sooner than in a conventional design.

The major performance gains are primarily due to macro-op fusing. These gains are not necessarily due to the specific types of instructions that are fused, rather they are due to the reduced number of separate instruction entities that must be processed by the pipeline. For example, two fused uops become a single instruction entity as far as the issue logic (and most other parts of the pipeline are concerned). Fusing 56% uops is similar (in terms of performance) to removing 28% *uops* without breaking correctness. Of course, other factors can affect the eventual performance gains related to fusing, for example, memory latency, branch mispredictions, etc.

In general, fused uops do not affect path lengths in the dataflow graph; however there are certain cases where they may increase or decrease path lengths (thereby adding or reducing latency). Cases of reduced latency (for some branches and loads) are mentioned above. Increased latency can occur, for example, 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 6~8% single-cycle ALU-ops that are not fused. Figure 6.8 also shows that our simple and fast runtime fusing heuristics may still cause slowdowns for benchmarks such as *parser* in SPEC2000 and *access* and *project* in WinStone2004. The speedup for a benchmark is mainly determined by its runtime characteristics and by how well the fusing heuristics work for it.

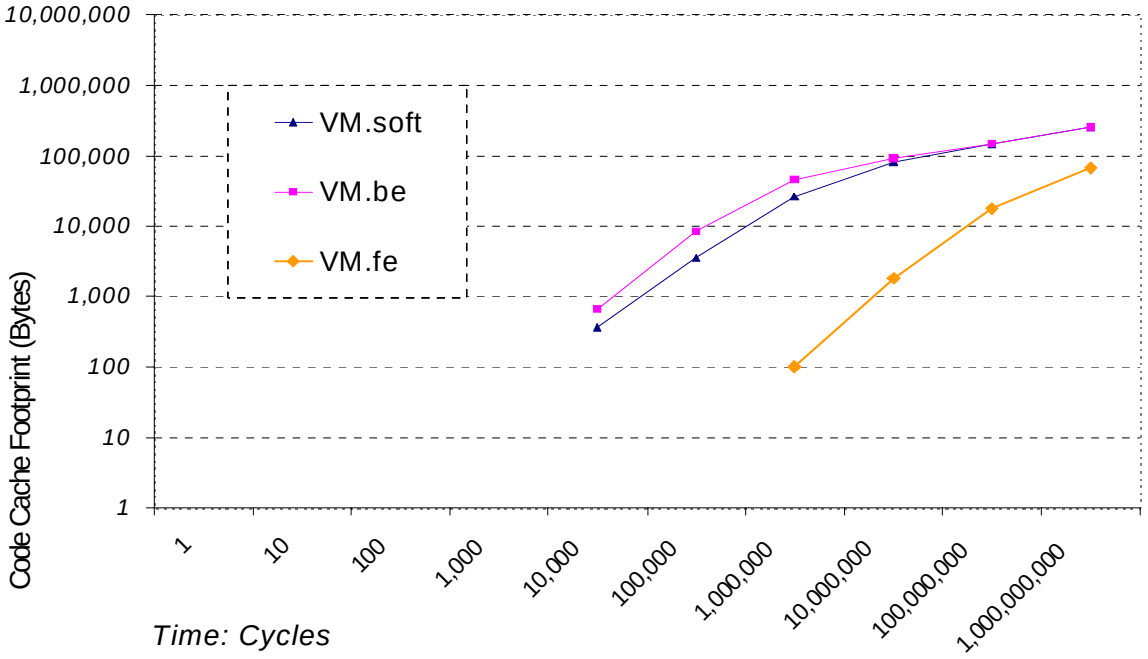
We also should make some observations for memory intensive benchmarks, particularly *mcf* and *gap* in SPEC2000. With our memory hierarchy configuration and the SPEC test input data set, there are 5.3 L2 cache misses per 1000 x86 instructions for the *mcf* x86 binary. This number is significantly lower than for the larger SPEC reference input data set. Hence, in our simulations, the poor memory performance typical of *mcf* does not overwhelm gains due to macro-op fusing as one might expect. On the other hand, the benchmark *gap* shows 19 L2 misses per 1000 x86 instructions, and performance improvements for *gap* are quite low.

Code Cache Footprint Analysis

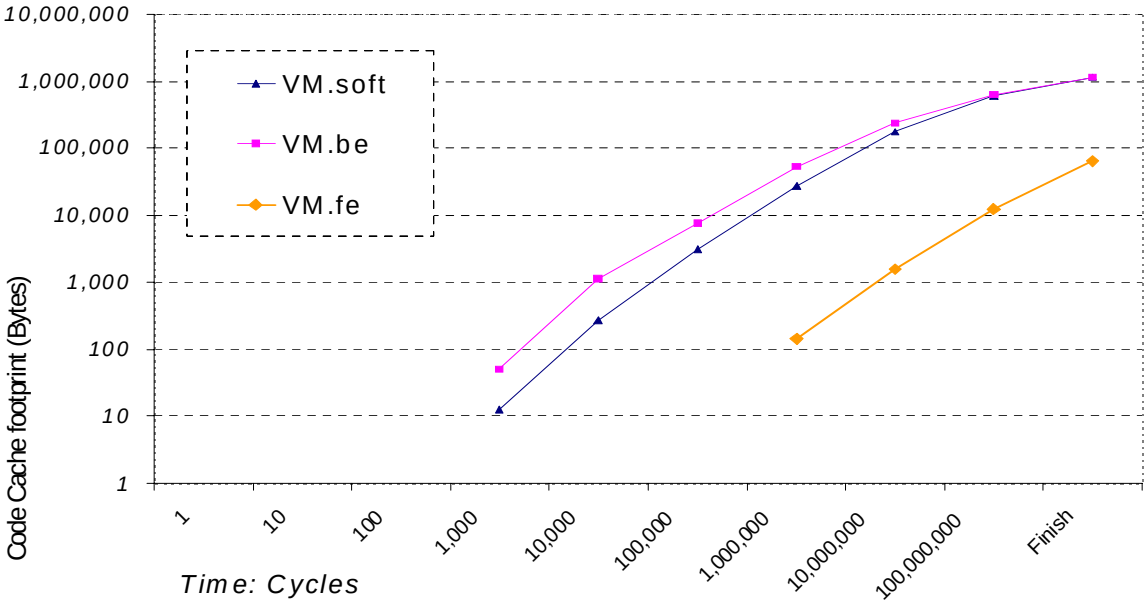
The co-designed x86 processor improves performance at the cost of some extra concealed physical memory space. The major part of the hidden memory area is allocated for the code cache that holds translated native code. Therefore, it is important to estimate the required code cache size. Such a characterization is more difficult than it may appear - it needs to track execution of trillions of instructions that run minutes or hours on real whole computer systems. With our infrastructure, we were only able to provide some evidence based on short simulations and it is shown in Figure 6.9.

Figure 6.9 shows how code cache footprints increase for the VM models listed in Table 5.4, Chapter 5. The x-axis shows the time in terms of cycles and it is on logarithmic scale. The y-axis shows code cache footprint in terms of bytes and it is also on logarithmic scale. All curves are averages over the benchmarks in the SPEC2000 and WinStone2004 suites respectively.

All models have increasing code cache footprint as workloads proceed. However, it is clear that for all models, the footprint increase slows down and begins flattening. Different VM models have different code cache footprints. A state-of-the-art VM such as the *VM.soft* model would take, on average, about 1MB for the 500M x86 instruction runs and 250KB for complete SPEC2000 test runs. On the other hand, because our example design (discussed in this chapter and labeled as *VM.fe* in the figure) has x86 mode to filter cold code, the co-designed x86 processor has a significantly smaller code cache footprint (0.1MB or less), only for hotspots. It is important to note that, based on Equation 2 in Chapter 3, the hotspot threshold is set at 4K for the SPEC2000 integer benchmarks and 8K for the WinStone2004 Business suite because they have different steady state IPC performance speedups, 18% versus 8% over the superscalar baseline.



(a) SPEC2000 integer benchmarks



(b) Windows benchmarks (WinStone2004 Business suites)

Figure 6.9 Code cache footprint of the co-designed x86 processors

Discussion

Without a detailed circuit level implementation of the proposed processor, some characteristics are hard to evaluate. For example, the potentially faster clock that results from pipelined issue logic and removal of the ALU-to-ALU bypass network.

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

6.4 Related Work on CISC (x86) Processor Design

Real world x86 processors

In virtually every design, decoder logic in high performance x86 implementations decomposes x86 instructions into one or more RISC-style micro-ops. The Cyrix 6x86 processor design [90] attempts to keep together all parts of an x86 instruction as it passes through its seven processing stages, though micro-ops are scheduled and executed separately. Some recent x86 implementations also have gone in the direction of more coarse-grained internal operations in certain pipeline stages. The AMD's K7/K8 microarchitecture [37, 74] maps x86 instructions to internal Macro-Operations that are designed to reduce the dynamic operation count. The front-end pipeline of the Intel Pentium M microarchitecture [51] fuses ALU operations with memory stores, and memory loads with ALU operations as specified in the original x86 instructions. The Pentium M processors also use a "stack engine" [16, 51] to optimize stack address calculations. However, the operations in each pair are still individually scheduled and executed in the core pipeline backend.

The fundamental difference between our fused macro-ops and the AMD and 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, including 2-cycle pipelined issue logic, collapsed 3-1 ALU(s) and a much simplified operand forwarding network; and (2) can be taken from different x86 instructions -- as our data shows, more than 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 x86 implementations maintain the fused operations for only part of the processor pipeline. For example, their individual micro-ops are scheduled separately by single-cycle atomic issue logic.

Macro-op execution

The macro-op execution microarchitecture evolved from prior work on coarse-grained instruction scheduling and execution [80, 81] and a dynamic binary translation approach for fusing dependent instruction pairs [62]. The work on coarse-grained scheduling [81] proposed hardware-based grouping of pairs of dependent RISC (Alpha) instructions into macro-ops to achieve pipelined instruction scheduling. The work on instruction fusing [62] proposed using single-pass fusing algorithm to efficiently fuse dependent micro-ops.

Compared with the hardware approach in [80, 81], we remove considerable complexity from the hardware pipeline and enable more sophisticated fusing heuristics, resulting in a larger number of fused macro-ops. Furthermore, in this thesis, we propose a new pipeline microarchitecture. For example, the front-end features a dual-mode x86 decoder, and the backend execution engine uniquely couples collapsed 3-1 ALUs with a 2-cycle pipelined macro-op scheduler to simplify the operand forwarding network.

Compared with the work in [62], I discovered a more advanced fusing algorithm than the single-pass algorithm in [62]. 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 I 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 other related research projects. Instruction-level distributed processing (ILDP) [77] 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 [110] use 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. It is evaluated with the MIPS-like PISA [23] ISA.

The IBM POWER4/5 processors also group five micro-ops (decoded and sometimes cracked from PowerPC instructions) into a single unit for the pipeline front-end only. The five micro-ops are close to a basic block granularity, and instruction tracking through the pipeline is greatly reduced.

The Dataflow Mini-Graph [20] 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 much more difficult (if it is even possible) for x86 binaries because variable length instructions and embedded data lead to extremely complex code “discovery” problems [61]. CCA, as proposed in [27], 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 [50] also collapses some instruction patterns. Continuous Optimization [44] and RENO [105] present novel dynamic optimizations at the rename stage. By completely removing some dynamic instructions (also performed in [45] 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 [2] is another hardware-based IA-32 dynamic optimization system capable of various optimizations.

Compared with these hardware-intensive optimizing schemes, our co-designed VM scheme strives for co-designed microarchitecture and software to reduce hardware complexity and power consumption. Perhaps more importantly, the co-designed VM paradigm enables more architecture innovations than hardware-only dynamic optimizers. Additionally, a software-based solution has more flexibility and scope when dealing with optimizations for a future novel architecture and subtle compatibility issues, especially involving traps [85].

Comparison of Co-designed virtual machine systems

Because the co-designed virtual machine paradigm is promising for future CISC x86 processor design, we compare several existing co-designed virtual machine systems, including the co-designed x86 virtual machines explored in this dissertation. Table 6.2 lists the comparisons for major VM aspects such as Architected ISA, implementation ISA, underlying microarchitecture and translation mechanisms.

It is clear from the table that different design goals result in different design trade-offs. For example, if competitive high performance (with conventional processor designs) for general-purpose computing is not the design goal, then startup performance is not critical. The removed hardware circuits help to reduce complexity and power consumption as exemplified by Transmeta x86 processor designs.

Table 6.2 Comparison of Co-Designed Virtual Machines

CO-DESIGNED VIRTUAL MACHINES	IBM AS/400	IBM DAISY/BOA	TRANSMETA CRUSOE / EFFICON	WISC ILDP	WISC X86VM
Design Goal	ISA Flexibility	High Performance Server Processor	Low Power and Low Complexity	Efficiency and Low Complexity	Design Paradigm for Efficiency
Architected ISA	Machine Interface	PowerPC	x86	Alpha	x86
Implementation ISA	CISC IMPI. Later: PowerPC	VLIW	VLIW	ILDP	Fusible ISA (RISC-style)
Microarchitecture	Superscalar	VLIW (8/6-wide)	VLIW (4/8-wide)	ILDP	Macro-op EXE
Initial Emulation	Program Translation is not transparent. It is performed at load time. Translated code can be persistent for later reuse.	Interpreter.	Interpreter / BBT	Interpreter	BBT or Dual-mode
Initial Emulation: HW Assist		No evidence	Efficeon: HW Assists interpreter	No	XLTx86 / Dual Mode Decoder
Hotspot Detection		Software	Software?	Software	Hardware
Hotspot Optimization		Software	Software	Software	Software
Hotspot Optimization HW Assist		No evidence	Unknown	No	Special new Instructions
Other unique features			HW support for speculative VLIW LD/ST reordering	HW Assist Indirect Control Transfer	

VLIW is selected as the execution engine for IBM DAISY/BOA and Transmeta co-designed x86 processors. However, all the other VM systems implement a superscalar or a similar microarchitecture (ILDP and macro-op execution) that is capable of dynamic instruction scheduling. This is especially important for today's unpredictable memory hierarchy behavior and its performance impact.

Chapter 7

Conclusions and Future Directions

The co-designed virtual machine paradigm has a long history dating back at least to the IBM System 38 [17] and AS/400 systems [12]. From its conception, the co-designed VM paradigm has been intended to handle machine interface complexities via its versatility. It has motivated pioneer projects from IBM, Transmeta and a few others. However, due to technical challenges and non-technical inertia, the application of this paradigm is still quite limited.

As the two fundamental computer architecture elements (ever-expanding applications and ever-evolving implementation technology) continue, perhaps it will no longer be possible to avoid the challenge of running a huge body of standard ISA software on future novel architectures. The co-designed VM paradigm carries the potential to mitigate such a fundamental tension for architecture innovations.

In this thesis, I have followed the pioneers' insight and have attempted to tackle the challenges they faced. The objective is a systematic research of the co-designed VM paradigm. During the thesis research, I conquered obstacles and made a number of discoveries. In this final chapter, I summarize the findings and conclude in Section 7.1. The encouraging conclusions suggest continuing research effort, and I discuss future directions in Section 7.2 which intends to justify investment in and application of the paradigm. Finally, Section 7.3 contains some reflections.

7.1 Research Summary and Conclusions

To evaluate a design paradigm for computer systems, the following three aspects are fundamental: capability, performance, and complexity or cost effectiveness. The entire thesis research is summarized and evaluated according to these three dimensions.

Capability

It is software that provides eventual solutions to computing problems. Therefore, *binary compatibility* practically implies capability, that is, the ability to run the huge body of available software distributed in standard binary formats, usually a widely used legacy ISA such as the x86. The pioneers from IBM and Transmeta have already proved the fundamental aspect that the hardware/ software co-designed VM paradigm can maintain 100% binary compatibility.

In this thesis, we did not extensively discuss this most important issue. However, in the experimental study, I did not encounter any significant obstacle regarding compatibility. Perhaps the real question boils down to emulation efficiency via ISA mapping. An informal discussion of this consideration is included in Section 7.3.

System *functionality integration and dynamic upgrading* is another manifestation of capability. In conventional processor designs, because of hardware complexity concerns, many desired functionalities are not integrated for practical reasons. For example, some of the attractive features include dynamic hardware resource management, runtime adaptation of critical system parameters, advanced power management and security checks, among many others. Furthermore, because a hardware bug cannot be fixed without calling back delivered products, there must be an exhaustive and expensive verification to guarantee 100% correctness for the integrated functionalities. Moreover, features implemented on a chip cannot be easily upgraded.

In contrast, the co-designed VM paradigm takes advantage of flexible and relatively cheap software to implement certain functionality; software components can be patched and upgraded at a much more affordable price. This practical functionality integration and dynamic update capability enables many desirable processor features regarding runtime code transformation and inspection for performance, security, reliability etc.

In this thesis, we implemented the x86 to fusible ISA mapping via co-designed software. The fusing algorithms are often beyond complexity-effective hardware design envelope. More runtime optimizations can be added, revised or enhanced later and the upgrade of the VM software is simply a firmware download similar to a BIOS update. Rigorous verification is still needed, but at a much lower cost than pure hardware requires.

Performance

After solutions become available, performance is usually the next concern. The eventual goal for the co-designed VM paradigm is to enable novel computer architectures without legacy concerns. Therefore, by its nature, the VM should have superior performance and efficiency at least in steady state, which is the dominant runtime phase for most applications. VM startup is affected the most by ISA translation overhead and consequently has long been a concern. This thesis work demonstrates that by combining balanced translation strategy, efficient translation software algorithms, and primitive hardware assists, VM startup behavior can be improved so that it is very competitive with conventional processor designs. Overall, the VM paradigm promises significant future performance potential.

In this thesis, I proposed an example co-designed x86 virtual machine that features a macro-op execution microarchitecture. One of the purposes of the co-designed x86 processor is to illustrate an efficient, high performance VM design. By fusing dependent instruction pairs, and processing fused

macro-ops throughout the entire pipeline, processor resources are better utilized and efficiency is improved. Fused macro-ops reduce instruction management and inter-instruction communication. Furthermore, fused macro-ops collapse the dynamic dataflow graph to better extract ILP. Overall, performance is improved via both higher ILP and faster clock speed. Primitive hardware assists, such as dual-mode decoders or translation-assist functional unit(s), ensure that the VM can catch up with conventional high performance superscalar designs during the program startup phase. After the DBT translator detects hotspots and fuses macro-ops for the macro-op execution engine, the VM steady state IPC performance improves by 18% and 8% over a comparable superscalar design for the benchmarked SPEC2000 and Windows workloads, respectively. Another significant performance boost comes from the faster clock speed potential that is enabled by the pipelined instruction scheduler and the simplified result forwarding network.

Complexity

Modern processors, especially high end designs, have become extremely complex. There are several consequent implications due to extreme complexity, i.e. longer time-to-market, higher power consumption and lower system reliability among many others.

The co-designed VM paradigm enables novel efficient system designs and shifts functionality to software components when appropriate. This approach fundamentally reduces overall complexity of a computer system. Less functionality implemented with transistors not only means reduced product design complexity, but also implies, if properly engineered, reduced power consumption and reduced probability for undetected reliability flaws.

In this thesis, the example co-designed processor also demonstrates complexity-effectiveness. From the pipeline front-end to backend: x86 decoders can be removed if a simplified backend functional unit is equipped for translation; the instruction scheduler logic can be pipelined for

simplified wakeup and select logic designs; the unique coupling of pipelined two-cycle scheduler with 3-1 collapsed ALU(s) removes the need for a complex ALU-to-ALU operand forwarding/bypass network. Additionally, if the design goal is to maintain a comparable performance to conventional designs, then the improved pipeline efficiency leads to a reduced pipeline width for better complexity effectiveness. The example VM design requires only minimal revisions to the current mature and well-proven superscalar designs. Therefore, the time-to-market and reliability should be able to maintain at least similar to current product levels. Power consumption should be able to decrease further as fused macro-ops can collect more efficiency benefits than Pentium M does from reduced instruction traffic through the pipeline.

In short, the co-designed VM paradigm provides more versatility and flexibility for processor designers. This flexibility advantage can be converted to capability, performance and complexity effectiveness advantages for future processor designs. In this thesis research, we demonstrate that the key enabling technology, efficient dynamic binary translation, can be achieved via sound engineering. Efficient DBT incurs acceptable overhead and enables the translated code to run efficiently. The combination of efficient DBT and a co-designed processor enable new efficient microarchitecture designs that are beyond the conventional microprocessor technology.

7.2 Future Research Directions

This thesis research confirms and provides further support for the co-designed VM paradigm. However, due to limited resources, it is not feasible for us to address all the details and explore all the possibilities. The thesis evaluation is also by no means exhaustive. Therefore, for a complete endeavor that explores this paradigm, we enumerate three important directions for future research and development efforts: confidence, enhancement and application.

Confidence

For real processor designs, broad range of benchmarks need to be tested and evaluated to achieve high confidence for any new system designs proposed. For example, a general-purpose processor design needs to evaluate all typical server, desktop, multimedia and graphics benchmarks. Different benchmarks stress different system features and dynamic behaviors.

This thesis research has conducted experimental studies on two important benchmark suites, the SPEC2000 integer and the WinStone2004 Business suites. Both are primary choices for evaluating the thesis research: the SPEC2000 applications mainly evaluate CPU pipelines while the WinStone2004 Windows benchmarks stress full system runtime behavior and code footprint. However, our SPEC2000 runs use test input datasets (except 253.perlbmk) and WinStone2004 runs are trace-driven short runs up to 500M x86 instructions.

For future work, we propose to improve confidence of the thesis conclusions via full benchmark runs over a more exhaustive set of benchmarks. For example, the effect of intensive context switches on code cache behavior over long runs is not clear with our evaluation. Such interactions might be important for servers under heavy workloads. Full benchmark runs with realistic input data sets can provide more valuable prediction for real system performance.

This is a more significant and challenging effort than it appears at a first glance. It requires a new experimental methodology that accurately evaluates full benchmark runs. There are related proposals for full benchmark simulations based on statistics, for example, SMARTS [128]. However, some workload characteristics and consequently their related system evaluations are based on runtime accumulated behavior. These evaluations involve some complex interactions that are still not clearly understood and may not be easily determined by statistical sampling. An example is the interaction between context switches and code cache behavior.

For the example co-designed x86 processor, there are also other ways to improve confidence of the conclusions. A circuit level model can provide solid evidence to verify the prediction that the macro-op execution engine can reduce power consumption and processor complexity. And it is also interesting to investigate how the macro-op execution performs on other major benchmarks such as server, embedded, mobile and entertainment workloads.

Enhancement

The co-designed VM paradigm itself is a huge design space. Consequently, we have explored only a small fraction of the space. The space we covered is guided by two heuristics: (1) address the critical issues for the paradigm in general, and (2) address the issues in a specific example of co-designed x86 virtual machine design. A valuable future direction is then to explore more of the design space for potential enhancements of the VM paradigm.

For the VM paradigm in general, there probably exist more synergetic, complexity-effective hardware/software co-designed techniques and solutions. For example, as the diversity of workloads increases, adaptive VM runtime strategies and runtime resource management also become more important. Benchmark studies revealed that the hotspot threshold still has room for improvement by exploring runtime adaptive settings.

For the co-designed x86 processor in specific, there are also many possible enhancements to be explored. The macro-op execution engine needs two register write ports per microarchitecture lane and the characterization data shows that only a small fraction of macro-ops actually need both ports. A valuable improvement is to reduce the register write port requirement. The hotspot optimizer can integrate more runtime optimizations that are cost-effective for runtime settings. Some possible optimizations include partial redundancy elimination, software x86 stack manager and SIMD-

ification [2] aforementioned. The fusible ISA might be enhanced by incorporating some new instructions that are essentially fused operations already in the first place instead of using a hint fusible bit.

Application

In this research, we show specifically how the macro-op execution architecture can be enabled by the co-designed VM paradigm. The co-designed x86 processor serves both as a research vehicle and a concrete application of the VM paradigm.

As a general design paradigm, a co-design VM can enable other novel architectures. However, the application of the VM paradigm to a specific architecture innovation will require further specific engineering effort to develop translation and probably other enabling technologies for a particular system design.

Additionally, many of the findings in the co-designed VM paradigm can be applied to other dynamic translation based system. For example, the primitive hardware assists could be deployed to accelerate runtime translation of other types of VMs. These VMs include system VMs that multiplex or partition computer systems at ISA level (e.g. VMware [126]) and process-level VMs that virtualize processes at ABI level (for example, various dynamic optimization systems, software porting utilities and high level language runtime systems).

7.3 Reflections

This dissertation has formally concluded – All previous thesis discussions are supposed to be based on scientific research methodology and solid experimental evidence. However, it is often enlightening to discuss issues *informally* from alternative perspectives. This additional section shares some informal opinions that I feel were helpful during my thesis research endeavor. However, these opinions are reflections -- they are subject to change.

- *An alternative perspective of co-designed virtual machines*

According to the classic computer architecture philosophy, a computer system should use hardware to implement primitives and use software to provide eventual solutions. This insight points out the overall direction for complexity-effective system designs for optimal benefit/cost ratio. Conventionally, processors are simply presumed to be such primitive-providers. However, modern processor designs are becoming so complex that they are complex systems themselves. Perhaps the better way to handle such complexity is to follow the classic wisdom – the processor should have its own primitives and solutions for hardware and software division.

In a sense, a major contribution of this work, simple hardware assists for DBT, is more or less a re-evaluation of current circumstances for processor designs according to this classic philosophy. Those proposed new instructions and assists are simply new primitives, and the hardware/software co-designed VM system is the solution for the entire processor design.

- *The scope of the Architected ISA*

The architected ISA is the standard binary format for commercial software distribution. As the x86 is more or less the *de facto* format, it might easily lead to a misconception that the architected ISA should be some legacy ISA. As a matter of fact, the architected ISA can be a new standard format as exemplified by Java bytecode [88] and Microsoft MSIL [87].

In fact, when the first co-designed VM came into being, the IBM AS/400 adopted an abstract ISA called MI (machine interface). All AS/400 software is deployed in the MI format. Dynamic binary translation generates the executable native code, which was initially a proprietary CISC ISA and later transparently migrated to the RISC PowerPC instruction set.

The scope of the architected ISA is not constrained to general purpose computing. Graphics and multimedia applications actually are distributed in one of the several standard abstract graphics binary formats, the architected ISA in the graphics/media world. Then, it is the graphics card device driver that performs the dynamic binary translation from the abstract graphics binary to the real graphics and media instructions that the specific graphics processor can execute. All ATI, *n*VIDIA and Intel graphics processors work in this paradigm. In a sense, the graphics/media processors (GPU) are probably the most widely deployed "co-designed virtual machines".

Although the GPU systems organize GPU cores and device driver very similarly, GPU vendors do not call their products co-designed virtual machines. And an interesting perspective is to consider the co-designed VM software to be a main processor device driver. However, the following two subtle differences (at least) differentiate them apart.

First, device drivers are managed by OS kernels that run on top of main CPU processors. These CPU processors execute the architected ISA of the system. In contrast, co-designed VM software is transparent to all conventional software, including OS kernel and device driver code. The co-designed VM is the system that implements and thus under the system architected ISA.

Second, GPU(s) are add-on devices that perform only the application-specific part of computation. Between scene switches, GPU drivers translate a small footprint of graphics code, all of which is hotspot code deserving optimizations. In contrast, the co-designed main processor conducts all the system functionalities. It has much larger code footprint and the code frequency varies dramatically.

❍ *The Architected ISA -- is x86 a good choice?*

The short answer is, a subset of the x86 is the best choice for an architected ISA so far. This is not only because more software has been developed for x86 than any other ISA, but also because there are important x86 advantages that are often overlooked for binary distribution.

Although communication bandwidth and storage capacity have been increasing dramatically, so have been the number of running software and software functionality, complexity and size. Hence, code density remains a significant factor. The x86 code density advantage not only comes from its variable instruction length, but also comes from its concise instruction semantics, for example, addressing modes.

There are unfortunate features in all ISA designs. This fact is probably unavoidable, especially from a long term historical perspective. Interestingly, the x86 dirty features tend to hurt less runtime behavior than those of RISC instruction sets.

For example, in RISC ISA designs, there are verbose address calculations, delayed branch slots and all kinds of encoding artifacts and inefficiency caused by 32-bit fixed length instructions (often overlooked for simple decoders). For all applications, many memory accesses and immediate values can be affected inherently by these encoding artifacts, there are few workarounds.

On the other hand, in the x86, there is a segmented memory model and a stack based x87 floating point that does not maintain precise exceptions. However, segmented memory is replaced by a page-based flat memory model and most x87 code can be replaced by SSE extensions - a better, parallel FP/media architecture that maintains precise exceptions. Amazingly, x86 applications often have some escape mechanisms to get away from the dirty and obsolete ISA features. In a sense, the x86 instruction set looks like an ISA framework that can accommodate many new features. Old, obsolete features can be replaced and eventually forgotten.

Perhaps a harder problem for x86 is the extra register spill code that is difficult to remove for dynamic binary translators. 100% binary compatibility requires 100% trap compatibility and memory access operations are more subtle to remove than it appears for such compatibility.

For ISA mapping, however, emulating a 16-register (or less) architecture (x86) on a 32-register processor is practically much easier and efficient than emulating an architecture with 32-register or more [89]. Although a co-designed processor can always have more registers in its implementation ISA than the architected ISA, more than 32 registers for an ISA design tends to be overkill and hurts code density [74].

- *ISA mapping -- the cost of 100% compatibility*

In theory, 100% binary compatibility can be maintained via ISA mapping. In practice, there is the problem of mapping efficiency. Here is an informal attempt to argue that such an ISA mapping is likely to be efficient for today's computer industry.

Intuitively, all computation can be boiled down to a Turing machine that processes a single bit at a time, and this simple time step consists of an atomic operation. At this level, all machines are proven to be equivalent for capability. The co-designed VM is essentially a machine model that has more states, but it has shorter critical path for common (hotspot) cases than conventional designs.

In reality, all processors employ a set of functional units to perform operations atomically at a higher level, such as ALU operations, memory accesses and branches. Mapping across different architectures at this level can be difficult. For example, mapping from the Intel x87 80-bit internal floating point to 64-bit floating point is both difficult and inefficient. The key point is that the co-designed VM can employ the same set of basic operations as the architected ISA specification. For

example, in our co-designed processor, the fusible ISA can employ the same basic floating point operations as in the x86 specification.

If all the basic ALU, memory, branch, FP/media and other operations can be matched at the functional unit level, then the ISA mapping is essentially cracking instructions into these atomic operations, and then combining them in a different way for the new architecture design. In fact, the abstract ISA interface inside our *x86vm* framework (between the top-level abstract *x86vmm* and *Microarchitecture* classes) is based on this set of atomic operations. In a sense, at least for common frequent instruction sequences, the ISA mapping is similar to an inexpensive *linear* transformation across different bases within the same linear space.

Bibliography

[1] Jaume Abella, Ramon Canal, Antonio Gonzales, “Power and Complexity-Aware Issue Queue Designs”, *IEEE Micro*, Vol. 23, No. 5, Sept. 2003.

[2] Y. Almog R. Rosner N. Schwartz, A. Schmorak. “Specialized Dynamic Optimizations for High Performance Energy-Efficient Microarchitecture”, *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, Mar. 2004.

[3] Erik R. Altman, *et al*. “BOA: the Architecture of a Binary Translation Processor”, *IBM Research Report* RC 21665, Dec 2000.

[4] Erik R. Altman, Kemal Ebcioglu, Michael Gschwind and Sumedh Sathaye, “Advances and Future Challenges in Binary Translation and Optimization”, *Proceedings Of the IEEE, Special Issue on Microprocessor Architecture and Compiler Technology*, Nov. 2001, pp. 1710-1722.

[5] Erik R. Altman, David Kaeli, and Yaron Sheffer, “Welcome to the opportunities of Binary translation”, *IEEE Computer* 33(3), Mar. 2000.

[6] AMD Corp., “AMD x86-64 Architecture Programmer’s Manual, vol. 1: Application Programming”, *AMD Corp.*, 2005

[7] AMD Corp., “AMD x86-64 Architecture Programmer’s Manual, vol. 2: System Programming”, *AMD Corp.*, 2005

[8] AMD Corp., “AMD x86-64 Architecture Programmer’s Manual, vol. 3: General Purpose and System Instructions”, *AMD Corp.*, 2005

[9] AMD Corp., “AMD x86-64 Architecture Programmer’s Manual, vol. 4: 128-bit Media Instrcutions”, *AMD Corp.*, 2005

[10] AMD Corp., “AMD x86-64 Architecture Programmer’s Manual, vol. 5: 64-bit Media and x87 Floating-Point Instrutions”, *AMD Corp.*, 2005

[11] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney, “Adaptive Optimization in the Jalapeno JVM”, *Proceedings of the ACM SIGPALN Conference on Object-Oriented Programming systems, Languages, and Applications*, pp. 47-65, Oct 2000.

[12] James E. Bahr, *et al*. "Architecture, Design, and Performance of Application Systems/400 (AS/400) Multiprocessors", *IBM Journal of Research and Development*, vol. 36, No. 1, pp. 12-23, Jan. 1990.

[13] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *Proceedings of International Symposium on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[14] Thomas Ball, James R. Larus. "Efficient Path Profiling". *Proceedings of the 29th International Symposium on Microarchitecture*, pages 46-57, 1996.

[15] Leonid Baraz, *et al*. "IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems" *Proceedings of the 36th International Symposium on Microarchitecture* pp. 191-202 Dec. 2003.

[16] Michael Bekerman, *et al*., "Early Load Address Resolution via Register Tracking", *Proceedings of the 27th International Symposium of Computer Architecture*, pp. 306-315, May 2000.

[17] V. Bertsis, "Security and Protection of Data in the IBM System/38", *Proceedings of the 7th International Symposium on Computer Architecture*, June. 1980, pp. 245-252

[18] Michael D. Bond, Kathryn S. McKinley, "Practical Path Profiling for Dynamic Optimizers", *Proceedings of the 3rd International Symposium on Code Generation and Optimizations*, pp. 205-216, Mar. 2005.

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

[20] Anne Bracy, Prashant Prahlad, Amir Roth, "Dataflow Mini-Graph: Amplifying Superscalar Capacity and Bandwidth", *Proceedings of the 37th International Symposium on Micro architecture*, Dec. 2004.

[21] Mary D. Brown, Jared Stark, and Yale N. Patt, "Select-Free Instruction Scheduling Logic", *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 204-213, Dec. 2001.

[22] Derek Bruening, Timothy Garnett, Saman Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization", *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pp. 265-275, Mar. 2003.

[23] 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.

[24] Anton Chernoff, Mark Herdeg, Ray Hookway, *et al*, "FX!32: A Profiler-Directed Binary Translator", *IEEE Micro* (18), March/April 1998.

[25] Yuan Chou, John P. Shen, "Instruction Path Coprocessors", *Proceedings of the 27th International Symposium on Computer Architecture*, Jun. 2000.

[26] Yuan Chou, Pazheni Pillai, Herman Schmit, John P. Shen, "PipeRench Implementation of the Instruction Path Coprocessor", *Proceedings of the 33rd International Symposium on Microarchitecture* pp. 147-258 Dec. 2000

[27] Nathan Clark, *et al*, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization", *Proceedings of the 37th International Symposium on Microarchitecture*, Dec. 2004.

[28] Robert F. Cmelik, David R. Ditzel, Edmond J. Kelly, Colin B. Hunter, *et al*, "Combining Hardware and Software to Provide an Improved Microprocessor", *US Patent 6,031,992*, Feb. 2000.

[29] Robert F. Cmelik, David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", *Technical Report UWCSE 93-06-06, University of Washington*, Jun. 1996.

[30] Robert Cohn, P. Geoffrey Lowney, "Hot Cold Optimization of Large Windows/NT Applications", *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 80-89 Dec. 1996.

[31] T. M. Conte, K. N. Menezes, M. A. Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer", *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996

[32] CRAY-1 S Series Hardware Reference Manual, Cray Research Inc., *Publication HR-808*, Chippewa Falls, WI, 1980.

[33] CRAY-2 Central Processor, Attempt: http://www.ece.wisc.edu/~jes/papers/cray2a.pdf *unpublished document*, circa 1979.

[34] CRAY-2 Hardware Reference Manual, Cray Research Inc., *Publication HR-2000*, Mendota Heights, MN, 1985.

[35] A. Cristal, *et al*., "Kilo-Instruction Processors: Overcoming the Memory Wall", *IEEE Micro*, pp. 48-57, May/Jume 2005

[36] James C. Dehnert, *et al*. "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges", *Proceedings of the 1st International Symposium on Code Generation and Optimizations*. Mar. 2003.

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

[38] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, George Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors", *Proceedings of the 27th International Symposium on computer Architecture*, pp. 316-325, Jun. 2000.

[39] Peter J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM* vol. 11 Number 5. May, 1968

[40] Ashutosh S. Dhodapkar, James E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," *Proceedings of the 29th International Symposium on Computer Architecture,* pp. 233-244, Jun. 2002

[41] Kemal Ebcioglu, Eric R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", *IBM Research Report RC 20538*, Aug. 1996 Also: *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[42] Kemal Ebcioglu, Erik R. Altman, *et al*., "Dynamic Binary Translation and Optimization", *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 529-548. June 2001.

[43] Joel S. Emer, Douglas W. Clark, "A Characterization of Processor Performance in the VAX-11 / 780", *Proceedings of the 11th International Symposium on Computer Architecture*, 1984.

[44] Brian Fahs, Todd Rafacz, Sanjay J. Patel, Steven S. Lumetta, "Continuous Optimization", *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[45] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. , Steven S. Lumetta, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 16-27 Dec. 2001.

[46] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. "The Multicluster Architecture: Reducing Cycle Time Through Partitioning." *Proceedings the 30th Iternational Symposium on Microarchitecture (MICRO-30)*, Dec. 1997

[47] John G. Favor, "RISC86 Instruction Set", *United States Patent 6.336,178*. Jan. 2002.

[48] E. Fetzer, J. Orton, "A Fully Bypassed 6-issue Integer Datapath and Register File on an Itanium-2 Microprocessor", *Proceedings of International Solid State Circuits Conference*, Nov. 2002.

[49] Joseph A. Fischer, "Very Long Instruction Word Architectures and the ELI-512", *Proceedings of the 10th International Symposium on Computer Architectures*, pp. 140-150, IEEE, June, 1983

[50] D. H. Friendly, S. J. Patel, Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", *Proceedings of the 31st International Symposium on Microarchitecture*, Dec. 1998.

[51] Simcha Gochamn *et al*., "The Intel Pentium M Processor: Microarchitecture and Performance", *Intel Technology Journal*, vol.7, issue 2, 2003.

[52] Michael Gschwind and Erik Altman, "Precise Exception Semantics in Dynamic Compilations", *Proceedings of 2002 Symposium On Compiler Construction*, pp. 95-110. April 2002

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

[54] Tom R. Halfhill, "Transmeta Breaks x86 Low-Power Barrier," *Microprocessor Report*, Feb. 14, 2000.

[55] Kim M. Hazelwood, Michael D. Smith, "Code Cache Management Schemes for Dynamic Optimizers", *Proceedings of the 6th Workshop on Interaction between Compilers and Computer Architecture*, pp. 92-100, Feb. 2002.

[56] John L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer,* Vol. 33, No. 7, pp. 28-35, Jul. 2000.

[57] Mark D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer*, Vol. 31, No. 8, Aug. 1998.

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

[59] Ron Ho, Kenneth W. Mai, Mark A. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, Vol. 89, No. 2, pp. 490-504, Apr. 2001

[60] Raymond J. Hookway, Mark A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation", *Digital Technical Journal*, vol. 9, No. 1, Jan. 1997.

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

[62] Shiliang Hu and James E. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions", *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pp. 213-224, Mar. 2004.

[63] Shiliang Hu, Ilhyun Kim, Mikko H. Lipasti, James E. Smith, "An Approach for Implementing Efficient Superscalar CISC Processors", *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pp. 40-51, Feb. 2006.

[64] Shiliang Hu, and James E. Smith, "Reducing Startup Time in Co-Designed Virtual Machines", *Proceedings of the 33rd International Symposium on Computer Architecture*. June, 2006.

[65] Wen-mei 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.

[66] IBM Corp., “The PowerPC Architecture”, *Morgan Kaufmann*, San Franciscon, 1994

[67] Intel Corp., “ IA-32 Intel Architecture Software Developer’s Manual, vol.1:Basic Architecture”, *Intel Corp.*, 2003

[68] Intel Corp., “ IA-32 Intel Architecture Software Developer’s Manual, vol.2: Instruction Set Reference”, *Intel Corp.*, 2003

[69] Intel Corp., “ IA-32 Intel Architecture Software Developer’s Manual, vol.3: System Programming Guide”, *Intel Corp.*, 2003

[70] Intel Corp., “ Intel Itanium Architecture Software Developer’s Manual” vol. 3, Instruction Set Reference, *Intel Corp.*, 2001.

[71] Quinn Jacobson and James E. Smith, “Instruction Pre-Processing in Trace Processors”, *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. 1999

[72] Rahul Joshi, Michael D. Bond, and Craig Zilles, “Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems”, *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, Mar. 2004.

[73] Tejas S. Karkhanis and James E. Smith, “A First-Order Superscalar Processor Model”, *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 338-349, June 2004

[74] C. N. Keltcher, *et al*., “The AMD Opteron Processor for Multiprocessor Servers”, *IEEE MICRO*, Mar.-Apr. 2003, pp. 66 -76.

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

[76] Ho-Seop Kim, “A Co-Designed Virtaul Machine for Instruction Level Distributed Processing”, *Ph.D. Thesis*, http://www.cs.wisc.edu/arch/uwarch/theses

[77] Ho-Seop Kim and James. E. Smith, “An Instruction Set and Microarchitecture for Instruction Level Distributed Processing”, *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 71-82, May 2002.

[78] Ho-Seop Kim and James. E. Smith, “Dynamic Binary Translation for Accumulator-Oriented Architectures”, *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pp. 25-35, Mar. 2003.

[79] Ho-Seop Kim and James. E. Smith, "Hardware Support for Control Transfers in Code Cache". *Proceedings of the 36th International Symposium on Microarchitecture,* pp. 253-264 Dec. 2003

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

[81] Ilhyun Kim and Mikko H. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints", *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 277-288, Dec. 2003.

[82] A. Klaiber, "The Technology Behind Crusoe Processors", *Transmeta Technical Brief*, 2000.

[83] K. Krewell, "Transmeta Gets More Efficeon", *Microprocessor Report*, vol.17, October, 2003.

[84] K. Lawton, "The BOCHS open source IA-32 Emulation Project", http://bochs.sourceforge.net

[85] Bich C. Le, "An Out-of-Order Execution Technique for Runtime Binary Translators", *Proceedings of the 8th International Symposium on Architecture Support for Programming Languages and Operating System*, pp. 151-158, Oct. 1998.

[86] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, Brian N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT", *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[87] S. Lidin, "Inside Microsoft .NET IL Assembler", *Microsoft Press*, Redmond, WA. 2002

[88] T. Lindholm, F. Yellin, "The Java Virtual Machine Specification", 2nd ed., *Addison-Wesley*, Reading, MA. 1999

[89] Jochen Liedtke, Nayeem Islam, Tent Jaeger, *et al*, "An Unconventional Proposal: Using the x86 Architecture as the Ubiquitous Standard Architecture", *Proceedings of the 8th ACM SIGOPS European Workshop on Support for composing distributed applications*, pp. 237-241, 1998.

[90] S. C. McMahan, M. Bluhm, R. A. Garibay, "6×86: the Cyrix solution to executing ×86 binaries on a high performance microprocessor", *Proceedings of the IEEE*, Vol. 83, Issue 12, pp 1664-1672, Dec. 1995.

[91] P. S. Magnusson, M. Christensson, J. Eskilson *et al*. "Simics: A Full System Simulation Platform", *IEEE Computer*, Vol. 35, issue. 2, pp. 50-58, Feb. 2002

[92] Nadeem Malik, Richard J. Elckemeyer, Stamatis Vassilladis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism", *ACM SIGMICRO Newsletter* Vol. 23, pp. 149–157, Dec. 1992

[93] C. May, "MIMIC: A Fast System/370 Simulator", *Proceedings of International Symposium on Programming Language Design and Implementation*, pp. 1-13, 1987.

[94] Steve Meloan, "The Java HotSpot Performance Engine: An In-Depth Look", *Technical Whitepaper, Sun Microsystems*, 1999.

[95] Stephen W. Melvin, Michael Shebanow, Yale N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines". *Proceedings of the 21st Annual Workshop and Symposium on Microprogramming and Microarchitecture*, pp. 60-63, 1988

[96] Matthew C. Merten *et al*, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," *Proceedings of the 26th International Symposium on Computer Architecture, May 1999*

[97] Matthew C. Merten *et al*, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots," *Proceedings of the 27th International Symposium on Computer Architecutre,* Jun. 2000

[98] Matthew C. Merten, Andrew R. Trick, Ronald D. Barnes, Erik M, Nystrom, Christopher N. George. John C. Gyllenhaal, Wen-mei W. Hwu, "An Architectural Framework for Runtime Optimization", *IEEE transactions on Computers*, Vol. 50, No.6 pp. 567-589, Jun, 2001.

[99] Pierre Michaud, Andre Seznec, "Dataflow Prescheduling for Large Instruction Windows in Out-oforder Processors," *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pp. 27-36, Jan. 2001.

[100] Gordon E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics,* Vol. 38, No. 8, pp. 114-117, Apr. 1965.

[101] Ravi Nair, M. E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups", *Proceedings of 24th International Symposium on Computer Architecture*, pp. 13-25, Jun, 1997

[102] S. Palacharla, N. P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors", *Proceedings of 24th International Symposium on Computer Architecture*, pp. 206-218, Jun, 1997

[103] David A. Patterson, Carlo H. Sequin: "RISC I: A Reduced Instruction Set VLSI Computer." *Proceedings of the 8th Internationall Symposium on Computer Architecture*, pp. 443-458, 1981

[104] S. J. Patel, S. S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization", *IEEE Transactions on Computers* (June), pp. 590-608. 2001.

[105] Vlad Petric, Tingting Sha, Amir Roth, "RENO – A Rename-based Instruction Optimizer", *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[106] 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.

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

[108] Mark E. Russinnovich and David A. Solomon, "Windows Internals", Fourth Edition, *Microsoft Press*, 2005

[109] K. Sankaralingam, *et al*. "Exploring ILP, TLP, DLP with the Polymorphous TRIPS Architecture", *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 422-433, May 2002

[110] Peter G. Sassone, D. Scott Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication", *Proceedings of the 37th International Symposium on Microarchitecture*, Dec. 2004.

[111] Y. Sazeides, S. Vassiliadis, J. E. Smith, "The Performance Potential of Data Dependence Speculation and Collapsing", *Proceedings of the 29th International Symposium on Micro architecture*, 1996

[112] Simulator: SimNow!, http://www.x86-64.org/downloads *AMD64 website.*

[113] R. Sites, A. Chernoff, Keik, M. Marks, and S. Robinson, "Binary Translation", *Communications of ACM* 36 (2), Feb. 1993, pp. 69-81.

[114] Brian Sletchta, *et al*. "Dynamic Optimization of Micro-Operations", *Proceedings of the 9th International Symposium on High Performance Computer Architecture*. Feb. 2003.

[115] James E. Smith and Ravi Nair, "Virtual Machines: Versatile Platforms for Systems and Processes", *Morgan Kaufman Publishers*, 2005.

[116] G. Sohi, S. Breach, T. Vijaykumar, "Multiscalar Processors", *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995

[117] G. Sohi, S. Vajapeyam, "Instruction Issue Logic for High Performance, Interruptable Pipelined Processors". *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 27-34, 1987.

[118] Jared Stark, Mary Brown and Yale Patt, "On Pipelining Dynamic Instruction Scheduling Logic", *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 57-66, Dec. 2000.

[119] E. P. Stritter, H. L. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor ", *Proceedings 11th Annual Microprogramming Workshop*, Nov. 1978, 8-16.

[120] J. M. Tendler, *et al*., "POWER4 System Microarchitecture", *IBM Journal of Research and Development*, Vol. 46. No. 1, 2002.

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

[122] Transmeta Corporation. Transmeta Efficeon Processor http://www.transmeta.com/efficeon/

[123] David Ung, Cristina Cifuentes, "Optimizing Hot Paths in a Dynamic Binary Translator", *Proceedings of the 2nd Workshop on Binary Translation*, Oct. 2000.

[124] David Ung, Cristina Cifuentes, "Machine-Adaptable Dynamic Binary Translation", *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pp. 41-51, 2000.

[125] K. Vaswani, M. Thazhuthaveetil, Y. N. Srikant, "A Programmable Hardware Path Profiler", *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March, 2005

[126] VMware Corp. "VMware Virual Platform Techinical White Paper", *VMware Inc.*, Palo Alto, CA, 2000

[127] Emmett Witchel, Mendel Rosenblum, "Embra: Fast and Flexible Machine Simulation", *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 68-78, 1996.

[128] Roland. E. Wunderlich, *et al*. "SMARTS: Accelerating Microarchitecture Simulation with Rigorous Sampling", *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 84-95, June, 2003

[129] Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessors," *IEEE Micro*, Vol. 16, No. 2, pp. 28-40, Mar. 1996.

[130] Cindy Zheng and Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation", *IEEE Computer* 33(3), pp. 47-52, March 2000.