# A CO-DESIGNED VIRTUAL MACHINE FOR INSTRUCTION-LEVEL DISTRIBUTED PROCESSING

by

Ho-Seop Kim

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN—MADISON

2004

# Abstract

A current trend in high-performance superscalar processors is toward simpler designs that attempt to strike a balance between clock frequency, instruction-level parallelism, and power consumption. To achieve this goal, the thesis research reported here advocates a microarchitecture and design paradigm that rely less on low-level speculation techniques and more on simpler, modular designs with distributed processing at the instruction level, i.e., *instruction-level distributed processing* (ILDP).

This thesis shows that designing a hardware/software *co-designed virtual machine* (VM) system using an accumulator-oriented instruction set architecture (ISA) and microarchitecture is a good approach for implementing complexity-effective, high-performance out-of-order superscalar machines. The following three key points support this conclusion:

- An accumulator-oriented instruction format and microarchitecture fit today's technology constraints better than conventional design approaches: The ILDP ISA format assigns temporary values that account for most of the register communication to a small number of accumulators. As a result, the complexity of the register file and associated hardware structures are greatly reduced. Furthermore, the dependence-oriented ILDP ISA format allows simple implementation of a complexity-effective distributed microarchitecture that is tolerant of global communication latencies.
- The accumulator-oriented instruction format and microarchitecture result in low-overhead dynamic binary translation (DBT): Because the underlying ILDP hardware provides a form of superscalar out-of-order processing, the dynamic binary translator does not need to perform aggressive optimizations often used in previous co-designed virtual machine

systems. As a result, the dynamic binary translation overhead is greatly reduced compared to these systems.

- The co-designed VM system for ILDP performs similarly to, or better than, conventional superscalar processors having similar pipeline depths while achieving lower complexity in key pipeline structures: This reduction of complexity can be exploited to achieve either a higher clock frequency or lower power consumption, or a combination of the two.

This thesis makes two main contributions. First, the major components of a co-designed VM for ILDP are fully developed: an accumulator-based ISA that is designed to support efficient dynamic binary translation; a complexity-effective accumulator-based distributed microarchitecture; a fast and efficient DBT mechanism and hardware-based control-transfer support mechanisms. Second, performance evaluations and complexity analysis support the key points of the thesis listed above. A sound evaluation methodology is established, including: a detailed, current-generation superscalar pipeline simulator that forces timing correctness by design, and a hybrid dynamic binary translator/timing simulator framework that allows simple and flexible analysis of a co-designed VM system.

# Acknowledgements

First and foremost, I would like to thank my wife, Sohyun Yu, for being always there. Without her, I would not have been able to endure the rigors of the Ph.D. study. My daughter, Hannah, has been the brightest part of my life and made me appreciate the true beauty of life. I would like to thank my parents, Sang Keun Kim and In Ja Lee for their constant support throughout my life. When I was struggling in the graduate school, I always thought what they would do if they were in my position. My sister Mi Yeon and brother Ho-Joon have been my best friends since we were little and I am deeply thankful that they have kept a good company to my parents while I was studying in Wisconsin. I am very grateful with my parents-in-law, Jang Hee Yu and Chung Ja Cho, who always treated me as a son. My brother-in-law, Jae Hoon, helped me many times when he was staying in the U.S.

Having the opportunity to do research with my advisor, Jim Smith, was one of the best things happened in my life. Not only is he one of the greatest minds in computer architecture, but he has also taught me so many things about research, writing, and life. I am grateful for his constant support and “loosely-coupled” research style (as Subbarao Palacharla put it) throughout my graduate tenure in Wisconsin. I am very thankful to the members of my Ph.D. committee, Jim Smith, Mikko Lipasti, Mike Schulte, Guri Sohi, and Charles Fischer for their helpful feedback on my thesis research. Throughout my future career, I hope I can meet the high standards set by the Wisconsin computer architecture faculties, Jim Goodman, Mark Hill, Mikko Lipasti, Jim Smith, Guri Sohi, and David Wood.

I would like to thank the students that shared the room 3652. Ashutosh Dhodapkar has been my best friend and discussion partner in the lab and has never failed to bring a little fun to the group. Jason Cantin and Tejas Karkhanis, who inhabited in the back-end of the room with us, provided

stimulating discussions and occasional fun activities. I have always looked up to Timothy Heil who never failed to show generous spirit by answering all of my (sometimes not the smartest) questions. Subbu Sastry helped me with a trace-driven simulator that put my research on ILDP on track. Marty Licht struggled with me in the early phase of the ILDP research. I wish the best luck to Shiliang Hu, Kyle Nesbit, Nidhi Aggarwal, and Wooseok Chang. I will miss the little get-togethers we had at Jim's house.

I thank the Department of Electrical Engineering staff for their administrative services. My special thanks go to Bruce Orchard who has always came through and fixed our computers so we can get things done. Kevin Lepak, Ilhyun Kim, and Trey Cain volunteered to administrate the x86 Condor clusters which proved be a great research asset.

This material is based on work supported by the National Science Foundation under Grant No. EIA-0071924, CCR-9900610, and CCR-0311361, the Semiconductor Research Corporation under Grant No. 2000-HJ-782 and 2001-HJ-902, Intel Corporation, and International Business Machines Corporation. I thank Konrad Lai at Intel for equipment support. I also thank Rabin Sugumar and Sharon Chuang for an internship opportunity at Sun Microsystems.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation, the Semiconductor Research Corporation, Intel Corporation, and International Business Machines Corporation.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

## 1.1 Background

Moore's prediction in 1965 [162] that the number of transistors per integrated circuit would double every couple of years has held true for almost four decades [163], to the point that it is considered an empirical law. For the more than three decades since the introduction of the first integrated circuit microprocessor (the Intel 4004 in 1971), ever-shrinking semiconductor technology has been the driving force behind continuous microprocessor performance improvements. Faster transistor switching speeds have allowed higher clock frequency designs while bigger transistor budgets have enabled more microarchitecture techniques to be used to increase instruction level parallelism (ILP).

Until recently, reducing the number of logic levels per clock cycle, i.e., increasing the pipeline depth [98][108][28] has been a popular technique for achieving higher clock frequencies than would be obtainable with technology scaling alone. Considering the long development cycle of a new generation design[1], the deeply pipelined design style was considered a desirable practice because it allows relatively easy clock frequency ramping, thus ensuring the longevity of the design [108][222].

There are negatives to this approach though, even neglecting the classic pipeline latch overhead problem [142]. First, to accommodate relatively increasing memory latencies, deeper pipeline buffers and more aggressive speculation techniques are required – which in turn make the

[1] In the reduced instruction set computer (RISC) heyday of the mid-80s, the objective was a new processor design every two years; now it takes from four to six.

design more complex and harder to validate [242]. Second, on-chip global wire latency has become relatively worse compared to transistor switching speed [4][30][110][154]. With deeper pipelining, the negative impact of the global wire latency can only get worse, further reducing the pipeline efficiency. Finally, power consumption increases with deeper pipelines, be it dynamic (from excessively high clock frequency) or static (from increased leakage current of low $V_T$ transistors) [101][170].

## 1.2 Instruction-Level Distributed Processing

As a consequence, an apparent microarchitecture trend is back toward *relatively simpler* designs [90] that try to strike a good balance between clock frequency, ILP, and power consumption. Nonetheless, even this approach faces technology difficulties, global wire latency for example, albeit to a lesser degree than with very highly pipelined and complex designs. For this reason, the research here advocates a design principle that relies less on low-level speculation techniques and more on simpler, modular designs with distributed processing at the instruction level, i.e., *instruction level distributed processing* (ILDP) [216].

To study the full potential of future ILDP architectures, the thesis research considers new instruction sets that are suitable for distributed microarchitectures. The goal is to avoid the encumbrances of instruction sets designed in a different era and with different constraints. A newly designed instruction format distinguishes global and local communication patterns more explicitly and works to reduce hardware complexity of most major pipeline structures. This leads to, in their simplest form, microarchitectures that are tolerant of interconnect delays, use a relatively small number of fast (high power-consumption) transistors, and support both high clock frequencies and relatively short pipelines.

There has been other research on new instruction formats and distributed microarchitectures [171][224][227] that include similar ILDP concepts such as:

- Considering microarchitecture as a distributed computing problem
- Accounting for communication as well as computation
- Localizing communication to small functional units while managing the overall structure for communication.

These earlier studies are mostly aimed at achieving higher performance by utilizing tens, if not hundreds, of distributed processing elements (PEs), especially for potentially high ILP workloads such as media processing. Unlike the other proposals, the proposed ILDP paradigm will work well with irregular programs, not just with high ILP programs.

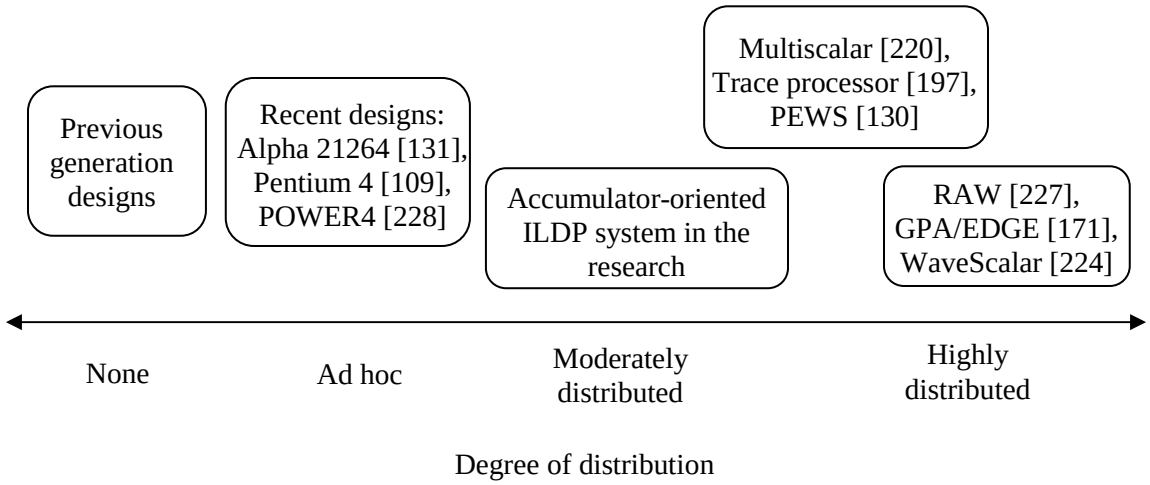



**Figure 1-1 Spectrum of distributed designs**

### 1.2.1 Accumulator-Oriented Instruction Set Architecture

The instruction set format used in the thesis is based on the following two rather fundamental observations. First, a large number of register values are used only once and most of them are used soon after they are created [86]. Second, instructions with true dependences cannot

execute in parallel (unless value speculation techniques [148][201] are used). If an instruction set architecture (ISA) is designed to explicitly convey dependence and register usage information, it would enable simpler implementations of dependence-based distributed microarchitectures that are complexity-effective and tolerant of global wire latencies.

The ILDP ISA studied in this thesis has a small number of accumulators backed with a relatively large number of general-purpose registers (GPRs). The instruction stream is divided into chains of dependent instructions, called *strands* hereafter, where intra-strand dependences are passed through a common accumulator. The general-purpose register file is used for communication between strands and for holding global values that have many consumers. Note that the ILDP ISA's emphasis on reduction of global *communications* by exploiting *dependences* between instructions is in stark contrast to the traditional reduced instruction set computer (RISC) ISAs where maximizing *computation* parallelism through *independences* between instructions has a high priority.[2]

### 1.2.2 Accumulator-Oriented Microarchitecture

The accumulator-oriented ISA in the thesis is specifically designed for an accompanying distributed microarchitecture. The overall ILDP microarchitecture shown in Figure 1-2 consists of pipelined instruction fetch, decode, and rename stages of modest width that feed a number of distributed processing elements, each of which performs sequential in-order instruction processing. The instruction set exposes inter-instruction dependences and local value communication patterns to the microarchitecture, which uses this information to steer chains of dependent instructions to the sequential PEs. Dependent instructions executed within the same PE have minimum communication

[2] In a sense, the role of accumulators (implying serial by default) in the dependence-oriented ILDP ISA can be thought as a dual of the stop bit (implying parallel unless stated otherwise) in the IA-64 architecture, a long instruction word (LIW) ISA designed for achieving high ILP [118].

delay as the results of one instruction are passed to the next through an accumulator. Taken collectively, the multiple sequential PEs implement multiple-issue out-of-order execution.

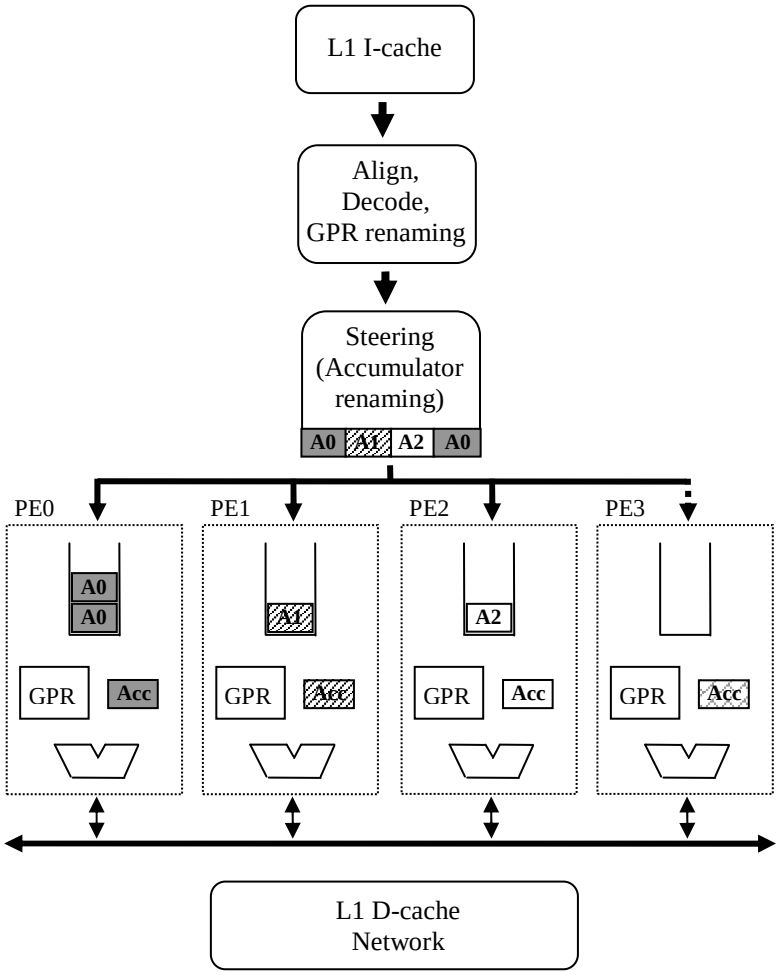



**Figure 1-2 High-level view of the ILDP pipeline**

## 1.3 Co-Designed Virtual Machine

### 1.3.1 Rationale

For certain applications where binary compatibility is not a major issue (e.g., in some embedded systems), a new instruction set may be used directly in its native form. However, for general-purpose applications a requirement of binary compatibility is a practical reality that must be dealt with. For these applications there are two possibilities, both involve dynamic binary translation from a virtual ISA (V-ISA) to an implementation ISA (I-ISA). One method is to perform on-the-fly hardware translation similar to the methods used today by Intel and AMD when they convert x86 instructions to micro-operations [63][81][98][108][209][210]. However, as will be shown in later sections, such a translation to an ILDP instruction set requires higher-level analysis than a simple instruction-by-instruction peephole mapping.[3] Hence, the second method relies on virtual machine software, co-designed[4] with the hardware and hidden from conventional software [217]. The binary translation subsystem in the virtual machine monitor (VMM) software layer maps existing binaries to the new ILDP instruction set in a manner similar in concept to the method used by the Transmeta Crusoe processor [60][99][128][140] and the IBM DAISY/BOA projects [9][68] [69][70][95]. All instructions are translated: applications, libraries, and the operating system –

---

[3] Technically, it is not impossible to implement the strand-oriented translation in a hardware-only manner, as was proposed in a recent study [199]. However, adding additional control hardware that works at a higher level than individual instructions is at odds with the overall goal of design simplification.

[4] IBM AS/400 series [12] (first introduced in 1988; now iSeries servers) are well-known co-designed virtual machine systems. The term, “co-design”, is also widely used in embedded design field [75]. As with the co-designed VM systems, the co-design methodology is used to achieve particular design goals.

everything. The translated codes are put into a special hidden memory area and are brought to the I-cache upon misses. Dynamic binary translation (DBT) can be performed either by a special co-processor [47][59] or by the main processor itself.

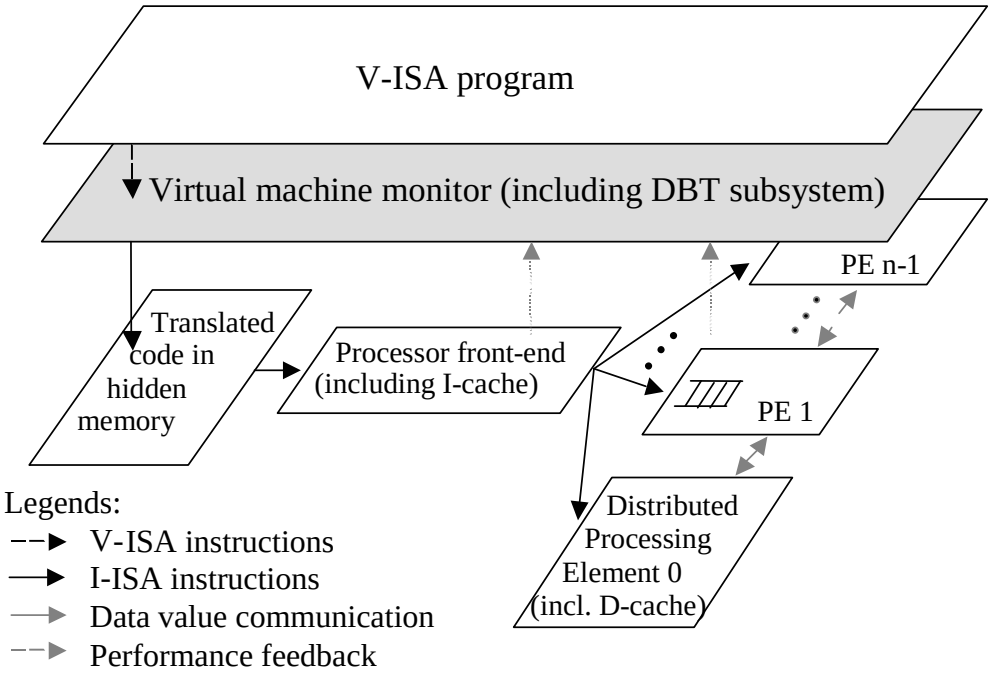



**Figure 1-3 Overview of the co-designed virtual machine in the thesis**

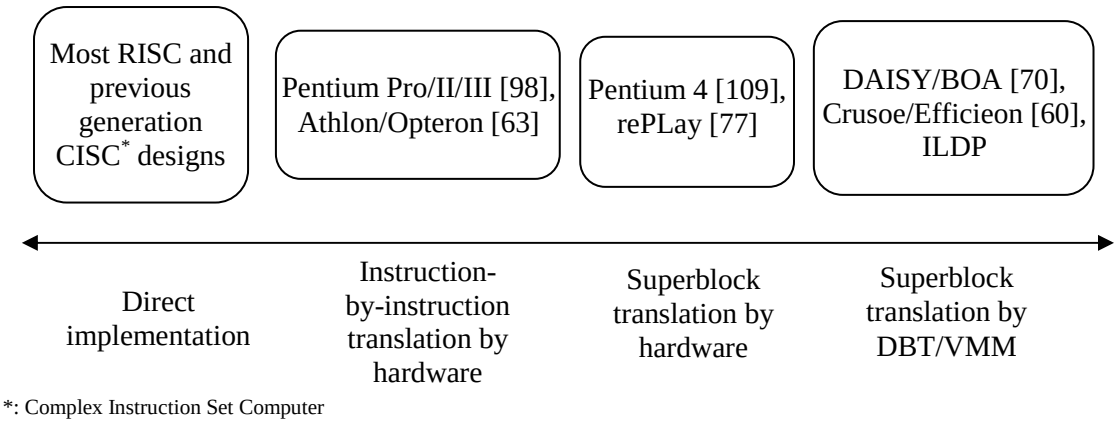



**Figure 1-4 Spectrum of dynamic translation mechanisms**

### 1.3.2 Dynamic Binary Translation

Dynamic binary translation converts instructions from a *source* ISA to a *target* ISA. In the co-designed VM paradigm, these are the V-ISA and I-ISA, respectively, and only the V-ISA is an existing instruction set for which conventional software exists. A DBT system also profiles program run-time behavior and dynamically optimizes blocks of frequently executed instructions.

The main objective of DBT in this research is identifying instruction inter-dependences and making register assignments that reduce intra-processor communication. This information is carried by the accumulator-oriented I-ISA and is used by the microarchitecture to steer instruction strands to the sequential PEs.

A key consideration in the design of a DBT system is the overhead resulting from the time it takes to translate; any time spent translating is time not spent executing the source program. In this thesis, the emphasis in the DBT system design is on *simplicity*; because the underlying hardware is a form of dynamic superscalar, it can be relied upon to provide code scheduling. An important difference from the two most notable co-designed VM systems, the IBM DAISY/BOA and the Transmeta Crusoe processor, is that here binary translation does not require complex optimizations and scheduling [71] that are used for the VLIW implementations in these systems. On the other hand, maintaining the original instruction order in the translated code helps maintain a precise trap recovery model [211].

### 1.3.3 Scope of Study

In this research, the Alpha instruction set is used as the outwardly visible V-ISA. By implementing all the major parts of a co-designed virtual machine, the research shows that the resulting ILDP system can achieve high performance through a combination of a high clock frequency, a moderate depth pipeline, and modest ILP. Simplicity is the key here – not only the

hardware complexities but also the dynamic binary translation overheads are reduced by designing the whole system around the accumulator-oriented instruction set and strand execution model. However, implementing a *fully*-working VM system is a huge engineering task and is beyond the scope of the thesis research. Instead, the thesis concentrates on (1) desired instruction set properties, (2) overall microarchitecture and key subsystem design, and (3) efficient dynamic binary translation support. The simplicity advantage will be demonstrated through the reduced complexities of the key pipeline hardware structures and a simple, fast dynamic binary translation algorithm, and instructions per cycle (IPC) performance is shown to be better or similar than comparable superscalar processors.

## 1.4 Thesis, Contributions, and Outline

The thesis supported by this research is that designing a hardware/software co-designed virtual machine system using an accumulator-oriented instruction set and microarchitecture is an effective approach for implementing complexity-effective, high-performance out-of-order superscalar processors. I defend this thesis by arguing the following three key points:

- *The accumulator-oriented instruction format and microarchitecture fit today's technology constraints better than conventional design approaches*: The ILDP ISA format assigns temporary values that account for most of the register values to a small number of accumulators. As a result, the complexity in the register file and associated hardware structures are greatly reduced. Furthermore, the dependence-oriented ILDP ISA format allows simple implementation of a complexity-effective distributed microarchitecture that is tolerant of global communication latencies.
- *The accumulator-oriented instruction format and microarchitecture results in low-overhead dynamic binary translation*: Because the underlying ILDP hardware provides a form of

superscalar out-of-order processing, the dynamic binary translator does not need to perform aggressive optimizations as used in previous co-designed VM systems. As a result, the dynamic binary translation overhead is greatly reduced compared to these systems.

- *The co-designed virtual machine system for instruction-level distributed processing performs similarly to or better than conventional superscalar processors of similar pipeline depth while achieving lower complexities in key pipeline structures*: This reduction of complexity can be exploited to achieve either higher clock frequency or lower power consumption, or a combination of the two.

My thesis makes two main contributions. First, I develop three key components of a co-designed virtual machine system for instruction-level distributed processing: an accumulator-based *instruction set architecture* designed to support efficient dynamic binary translation, a complexity-effective accumulator-based distributed *microarchitecture*, and a fast and efficient *dynamic binary translation* mechanism and hardware-based mechanisms for supporting control-transfers.

Second, I present performance evaluations and complexity analysis of the co-designed virtual machine system to illustrate its benefits and provide support for its use in future processor designs. With respect to the second contribution, a rigorous evaluation methodology was established including: a detailed, current-generation superscalar pipeline simulator with emphasis on correct timing and a hybrid dynamic binary translator/timing simulator framework that allows simple and flexible analysis of a co-designed VM system.

The thesis is organized as follows. The ILDP instruction set format is central to the thesis and is described in Chapter 2. First, a strand execution model is developed based on the observations on typical compiled code characteristics. After DBT-specific ISA requirements are discussed, the implementation-specific ILDP ISA formats are presented. Chapter 3 starts with an

overview of the accompanying ILDP microarchitecture and compares its unique operand capture [5] model to two prevalent superscalar models. Specifics of the ILDP microarchitecture are described afterwards. The dynamic binary translation mechanism, a key component of the virtual machine monitor layer, is explained in Chapter 4. The chosen unit of translation, the superblock (a straight-line code sequence with a single entry point and multiple exit points), and its formation rules are explained and then followed by discussions of two key issues in any DBT system: precise state reconstruction and dynamic code expansion. The actual DBT algorithm for the ILDP I-ISA is presented at the end of the chapter. Efficient support mechanisms for translating control transfer instructions are crucial to the performance of any code cache system[6]. For this reason, the topic is given a separate treatment in Chapter 5. In this chapter, problems with conventional superblock chaining mechanisms are identified. Combinations of specialized software/hardware mechanisms are proposed and evaluated separately from the rest of the thesis using an identity translation framework. This is to help understand the effect of the dynamic code re-layout that comes free with any superblock-based code cache system, separated from the ISA translation effect. Chapter 6 describes the experimental framework used in the thesis. This chapter not only discusses the modeling methodology of the ILDP co-designed VM system but also gives a detailed description of

---

[5] In modern pipelined designs, operand values can be obtained by accessing the designated data storage (e.g., the physical register file) in a pipeline stage or by constantly monitoring the bypass network. In the thesis, the term operand capture is used to represent both methods.

[6] Most dynamic translators and optimizers place frequently executed translated/optimized codes in a main memory area. Typically the term translation cache [70] is used for this memory area in ISA-translation systems while the term fragment cache [15] is used in optimization-only systems. Throughout the thesis, “code cache” is used as a general term for both translation cache and fragment cache. Because the ILDP VM system involves ISA translation, the term “translation cache” is sometimes used as well.

the baseline superscalar processor model. Some important complexity-effective design choices commonly employed in the current generation superscalar processor designs are also described for complexity comparisons. The evaluation of the ILDP system is presented in Chapter 7. Here, evaluation of the baseline superscalar processor model is first performed. These experiments serve two purposes. First the pipeline model, shared with the proposed ILDP microarchitecture, is validated. Second, the experiments help in understanding the performance bottlenecks in current-generation processor designs. The performance evaluation of the ILDP system consists of two main components; DBT-related characteristics and the overall IPC performance compared to the baseline superscalar pipeline. The simplicity advantages of the ILDP co-designed VM paradigm are showcased via complexity comparisons with respect to the conventional baseline microarchitecture. Emphasis is placed on reduced complexity of key pipeline structures and reduced translation overheads. Finally Chapter 8 summarizes the research and concludes the thesis. Further research opportunities are suggested at the end.

The thesis covers many different aspects of processor design, so a single chapter is not adequate to present all related work. Therefore each chapter, other than the final two, contains a related work section that is specific to the topics discussed in the chapter.

# Chapter 2 ILDP Instruction Set Architecture

The accumulator-oriented ILDP instruction set architecture is the centerpiece of this thesis research. Freed from the features of legacy ISAs introduced many decades ago, the thesis uses an instruction format that is well suited for today's technology constraints (thereby facilitating efficient hardware implementations) and yet it is a simple target for a dynamic binary translation system.

In this chapter, temporal locality of register values and inter-instruction dependences, two of the most fundamental properties of compiled programs, are re-examined in the context of today's technology constraints. The combination of these two properties leads to a strand-oriented execution model and a dependence-based, accumulator-oriented instruction format. A short code snippet, extracted from a benchmark program, is used to illustrate the overall idea of the execution model and the instruction format. Some of the representative characteristics of the chosen V-ISA that are relevant to the research are presented, and this is followed by discussions of precise state maintenance and dynamic code expansion in the context of dynamic binary translation. The ILDP ISA formats are designed to handle both of these issues well. The chapter ends with descriptions of the ILDP ISA formats in tabular form. Related execution paradigms and ISA ideas are listed at the end of the chapter.

## 2.1 Motivation

### 2.1.1 Separate Register Sets Based on Usage Patterns

Most existing ISAs do not distinguish between register value *usage* types and have only one level of register hierarchy. This simple flat model made good sense when ISAs were developed

decades ago [178]. On-chip registers were supposed to be small and fast enough for the intended level of pipelined designs at the time.

Today's typical high-performance superscalar processor designs [206][215] put lot of pressure on register files and associated structures. Dynamic register renaming (to remove false dependencies; further explained in section 3.1.3) requires physical register files to have more elements than architected, i.e., logical registers. Superscalar execution requires more access ports to the register file. Deeper buffering within the pipeline exacerbates these difficulties further because more physical register file entries are needed to support increased numbers of in-flight instructions. All in all, today's on-chip register file have become bigger and relatively slower.[7]

Reducing the pressure on the register file and associated hardware structures being one goal, the thesis research started with an observation that there is high locality in register usage patterns in most programs. It is well known that a large number of register values are used only once and most of them are used soon after they are created [86]. If these temporary register values can be offloaded to separate "scratchpad" registers, even a small number of registers will be sufficient for most instructions because the volatile scratchpad registers may be recycled quickly. Remaining register values that have long lifetimes can then be kept in another set of "backup" registers, most probably larger in number than the volatile scratchpad registers, but still less than in equivalent conventional designs. Therefore, this type of register organization will effectively reduce complexity pressure on both the number of ports *and* entries of the physical register file. Furthermore, the register renaming

[7] Although the physical register file itself can be pipelined to accommodate the increased access latency using more internal bypasses, it leads to increased shadow cycles as will be shown in Figure 3-3, reducing the effective issue window size and wasting energy in case of latency mis-speculation [138][164].

bandwidth requirement is also effectively reduced because fewer register instances are now renamed to the non-volatile register file.

### 2.1.2 Dependence-Oriented Program Execution

In general, instructions with true dependences cannot execute in parallel. Traditionally this fundamental property has been considered a hindrance to higher performance through ILP. An advantage of the reduced instruction set computer (RISC) paradigm is that a finer granularity ISA allows for increased parallelism between simpler instructions [229], through static scheduling by a compiler or dynamic scheduling by out-of-order execution hardware mechanisms. Prevalence of this idea led some implementations of complex ISAs, such as x86 to adopt a dynamic decomposition technique that transforms a complex external ISA instruction to multiple, simpler internal format instructions, sometimes called micro-operations [63][98][108][209][210].

One disadvantage of the fine-grain instruction execution model is that it tends to increase complexity pressure on the pipeline [137]. For example, even a temporary value that is used once is assigned a register entry and needs to be communicated from the producer to the consumer operation through the shared bypass network. As a reaction, some newer designs try to suppress the expansion of a complex instruction to a minimum number of pipeline stages [63][129], while others go as far as re-combining multiple dependent finer-granularity instructions into a coarse grain instruction [90][116][137]. One common idea behind these newer approaches is that if dependences are unavoidable, why not *exploit* them to reduce pipeline complexities?

The dependence-oriented microarchitecture studied by Palacharla, Jouppi, and Smith [175] employs dependence relationships to simplify the complexity-critical instruction issue pipeline stage. In their work, the traditional out-of-order issue window is replaced with multiple first-in, first-out (FIFO) buffers. Dependent instructions are steered to the same FIFO and only the

instructions at the FIFO heads need to be checked for issue. If the processor back-end functional units are clustered based on the FIFOs, most register value communication will happen within the same cluster and the frequency of long latency inter-cluster communications will be kept low.

### 2.1.3 Accumulator-Oriented Instruction Format Fits the Bill

If instruction sets were designed to convey dependence and register usage information more explicitly, it would be much simpler to implement dependence-based distributed microarchitectures that are complexity-effective and tolerant of global wire latencies. In other words, today's technology constraints would be better served with instruction sets that are designed according to the following principles:

- Provide separate register sets for temporary values which are used a small number of times and for values that have long lifetimes or many users.
- Provide a way of explicitly expressing inter-instruction dependences and their lifetimes.

An instruction set with a small number of accumulators and a relatively large number of general purpose registers (GPRs) fits these principles nicely. Chains of dependent instructions, *strands*, are associated with the same accumulator. After the first instruction in a strand writes to a given accumulator, each subsequent instruction reads and overwrites the same accumulator with a newly produced value for use by the next instruction in the strand. Values that have long lifetimes or are used multiple times are assigned to general purpose registers. In a distributed microarchitecture, an entire strand can be steered to the same cluster and instructions in the strand are then issued in order. A new strand is steered to a non-empty FIFO buffer if the last instruction in the FIFO was explicitly marked as an end-of-strand (i.e., it is now safe to overwrite previous strand's accumulator). An important requirement is that the instruction set semantics should be close enough to a traditional ISA to facilitate efficient dynamic binary translation.

### 2.1.4 Observed Program Characteristics

With an eye toward accumulator-based instruction sets, the thesis research started with manual inspection of frequently executed code sequences from the SPEC CPU2000 integer benchmarks [105] compiled for Alpha EV6 ISA. Soon it became evident that the dependence relationships between register value producers and consumers are not as complex as a typical RISC ISA may imply (with two input and one output registers). A trace-driven functional simulator was developed to quantify this observation. Alpha NOPs were removed from the trace before collecting the statistics in Table 2-1.

**Table 2-1 Alpha ISA program characteristics**

| Benchmark | Alpha instruction count | % of instructions with zero or one input register operand | % of loads without immediate | % of stores without immediate |
|---|---|---|---|---|
| *164.gzip* | 3.25 billion | 55.09 | 43.6 | 50.6 |
| *175.vpr* | 1.47 billion | 51.16 | 34.9 | 29.1 |
| *176.gcc* | 1.77 billion | 62.38 | 34.8 | 15.8 |
| *181.mcf* | 210 million | 57.74 | 30.4 | 11.9 |
| *186.crafty* | 4.07 billion | 54.34 | 27.0 | 13.4 |
| *197.parser* | 3.92 billion | 57.68 | 44.8 | 22.2 |
| *252.eon* | 89.7 million | 55.83 | 15.7 | 15.4 |
| *254.gap* | 1.11 billion | 61.60 | 44.9 | 27.1 |
| *300.twolf* | 238 million | 50.48 | 41.5 | 31.2 |

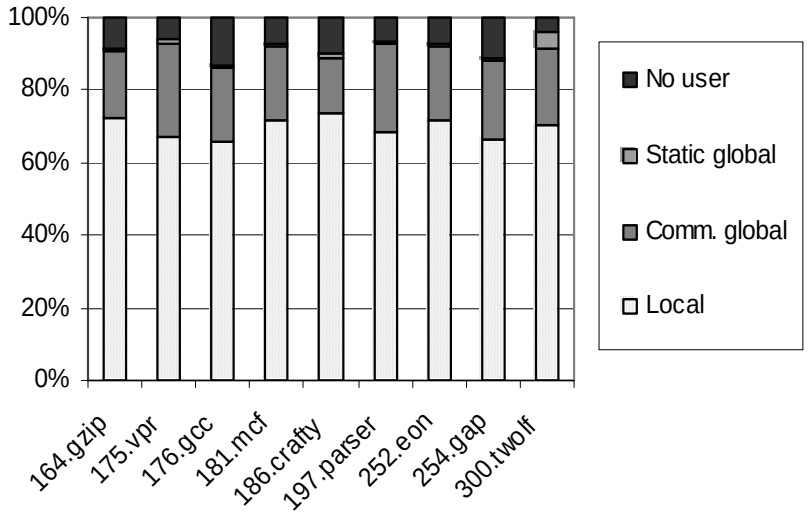
The fraction of instructions that have zero or one source register operand ranges between 50 to 62%. Loads, branches, jumps, and integer instructions with an immediate operand belong to this category. This suggests that many instructions are members of rather “thin” chains of dependent instructions, and there are relatively few dependence intersection points in programs.

Another interesting program characteristic is that there are a substantial number of memory access instructions that do not require address calculation. These instructions account for 15.7 to 44.9% for loads, 11.9 to 50.6 % for stores. Therefore, it can be inferred that a memory instruction format without address calculation has a good potential for lowering the average L1 D-cache load-to-use latency by bypassing the address adder. IA-64 architecture, a relatively new ISA (introduced in early 1990s), employs this feature [118].

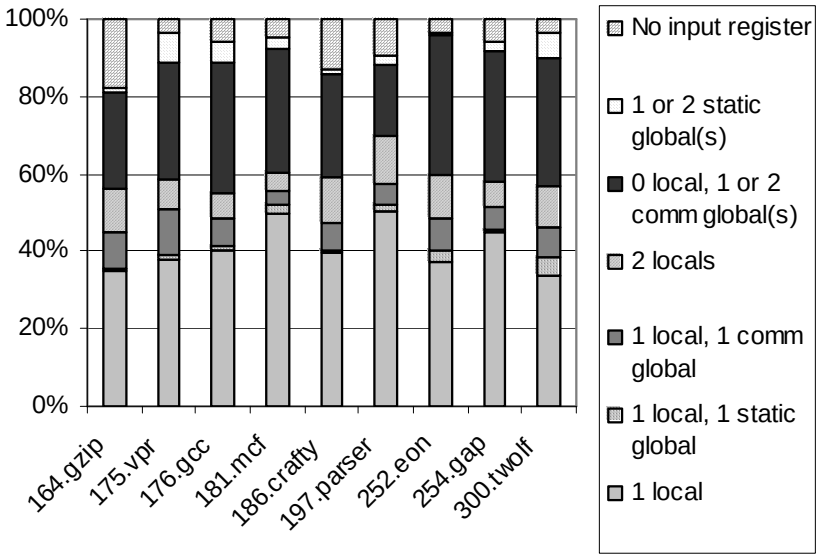
To further understand register usage patterns, register values are classified as follows:

- ***Local***: register values that are used only once
- ***No-user***: produced but not used by any instruction
- ***Static global***: used multiple times by the same static instruction, e.g., loop-invariant values
- ***Communication global***: used multiple times but not by the same static instruction

Local values are natural targets for accumulators that are accessed only locally inside a processing element. No-user values come from high-level program semantics; for example, a function's return value or some of its input arguments might not be used depending on the context. Also aggressive compiler optimizations, e.g., hoisting instructions above conditional branches, sometimes result in unused values [36]. Regarding global values, it is important to understand the difference between static and communication global values for distributed microarchitectures. In general, static global values tend to have long lifetimes and large number of consumers. These values are almost always immediately available from the register file when they are needed. On the other hand, communication global values tend to have relatively short lifetimes and smaller numbers of consumers compared to static global values. They are usually used fairly soon after being produced and hence, can have exposed inter-cluster communication latency if their producers and users are in different clusters.




(a) Output register value




(b) Input register value

**Figure 2-1 Type of register values**

Figure 2-1 shows the fraction of register values for each of the aforementioned classes. First, in Figure 2-1(a) about 70% of all produced values are local, confirming similar studies by others [37][86]. Instructions that do not produce register values, such as stores and conditional branches,

are not included. Figure 2-1(a) also shows that only about 20% of the produced values should be placed in global registers (which suggests relatively low global register write bandwidth).

The input register value profile shown in Figure 2-1(b) helps quantify the complexity of register dependence graphs in the collected program traces. The constant zero register (R31 in Alpha ISA) is treated as an immediate value and hence, is not counted as a register. The top two categories, where instructions do not have communication global or local values as input, do not affect overall performance (assuming static global values are always available immediately). Here, the Alpha load immediate instructions (LDA/LDAH) with the constant zero register as an input (similar to MIPS LUI instruction) and unconditional branch instructions (BR/BSR) are examples of the no input register category. The bottom two categories which have one local and zero communication global input value will not see any global communication latencies either. It is the three remaining categories in the middle that have a potential for having exposed global communication latencies due to communication global values. Note that in the fourth category (two locals), it is necessary to consider one of the two local input values as communication global – a dependence intersection point.

Figure 2-1(b) is pessimistic, however, in the sense that it considers all uses of a produced communication global value as communication global. When strands are identified, the first consuming instruction of a produced communication global value is allowed to continue the strand. This consumer instruction reads the value from the local accumulator. Another important point is that only those communication global values that happen to be on the program critical path might affect the program execution cycles [82]. That is, many register value communications can be hidden by parallel execution of multiple strands.

## 2.2 Strand: A Single Chain of Dependent Instructions

A strand is a *single* chain of dependent instructions and identifies the dependence chain's *lifetime*. The most important aspect of strand formation is the separation of local and global values because, in effect, it drives strand formation itself. When a source program is directly compiled to an accumulator-oriented ISA, a (static) compiler will perform dependence lifetime analysis and determine which values can safely be considered as local. In a dynamic binary translation system, the V-ISA (Alpha in the thesis) register usage patterns are first collected over a given translation unit before strand identification and accumulator assignments are performed.

### 2.2.1 Strand Formation

Two strands, carried by accumulators, denoted as An and Ak respectively, are shown in Figure 2-2. Each node represents a single instruction. A value that is used by both strands is also put in a GPR (denoted as Rj in the figure).

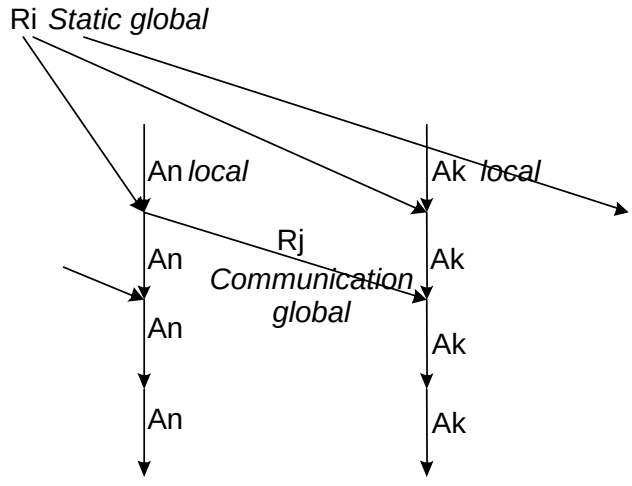



**Figure 2-2 Strand formation based on register value classification**

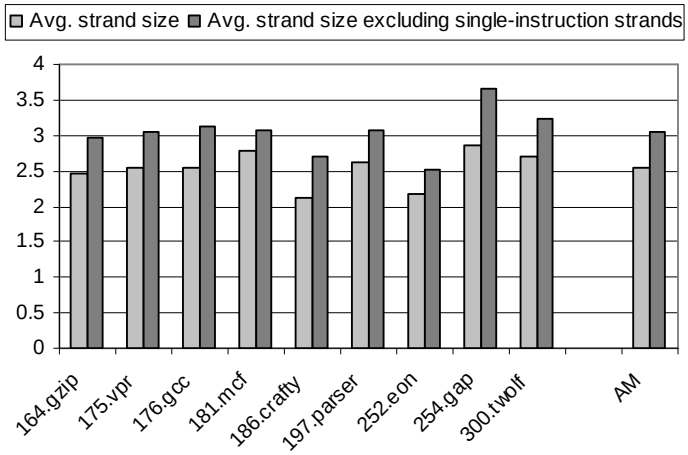
A strand is formed by the following rules:

- A strand is started with an instruction that has no local input values
- An instruction that has a single local input value assigned to a strand is added to the producer's strand
- If an instruction has two local input values, then two strands are intersecting. One of the strands has its local value converted to a communication global
- There is only one user of a produced value inside a strand
- Once all strands are identified within the given compilation/translation unit, the last instructions in each strand are marked as end-of-strand.

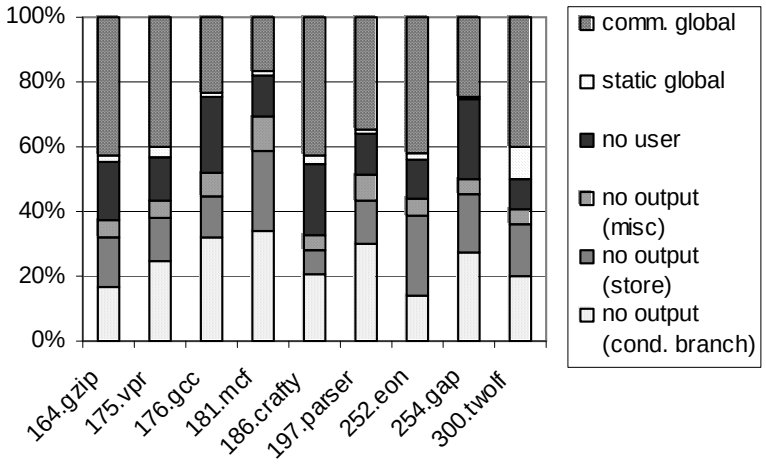
### 2.2.2 Strand Characteristics

Figure 2-3 shows two of the most important strand characteristics from the program traces. In Figure 2-3(a), average strand size is shown to be 2.54 Alpha instructions. Note that there are many single-instruction strands that do not contribute much to the total number of instructions but affect the average strand size significantly. These single-instruction strands include unconditional branch and jump instructions, and instructions whose produced values are not used.[8] If the single-instruction strands are ignored, the average size of strands is 3.04. It is also interesting to see how strands end (Figure 2-3(b)). About 35 to 70% of the strands end with conditional branch resolution and store address/value calculation. About 20 to 45% produce communication global values.

[8] An important characteristic of the single-instruction strands is that by definition, they start and end a strand at the same time. As a consequence, they can be steered to any available FIFO and any subsequent instructions can be steered to the same FIFO at the same cycle.




(a) Average strand lengths in Alpha ISA programs




(b) Strand end profiles

**Figure 2-3 Strand characteristics**

### 2.2.3 Dependence Lifetime vs. Variable Lifetime

There are only a finite number of accumulators available in a given instruction format. Therefore it is necessary to allocate accumulators to the identified strands. If the register allocator runs out of free accumulators, a strand is chosen to be terminated and its accumulator is “spilled” to

a general register. This can be compared to the traditional register allocation where a *variable*'s lifetime is assigned to register(s) [169]. Here a *dependence*'s lifetime is assigned to an accumulator.

There will be certain differences in optimization strategies between an ILDP ISA compiler and the traditional RISC compilers such as the one used to generate the Alpha binaries used in the thesis. Most optimizing RISC compilers try to increase parallelism between instructions by applying aggressive instruction scheduling techniques at the expense of increased register pressure. Too aggressive instruction scheduling, however, may not be beneficial in the accumulator ISAs because these parallel strands increase the accumulator pressure [92]. For that reason, it is probably better to rely on the underlying out-of-order superscalar hardware to achieve efficient code scheduling than forcing an ILDP compiler to schedule instructions aggressively.

### 2.2.4 Strand Execution Model

Figure 2-4 shows an Alpha code sequence dynamically translated into the accumulator-oriented ILDP ISA. It can be seen that there are four strands running in the code snippet, ranging from one to five Alpha instructions[9]. Local register values are shown in bold letters in Figure 2-4(b). Other values are global. A dotted arrow represents a value communication between two strands.

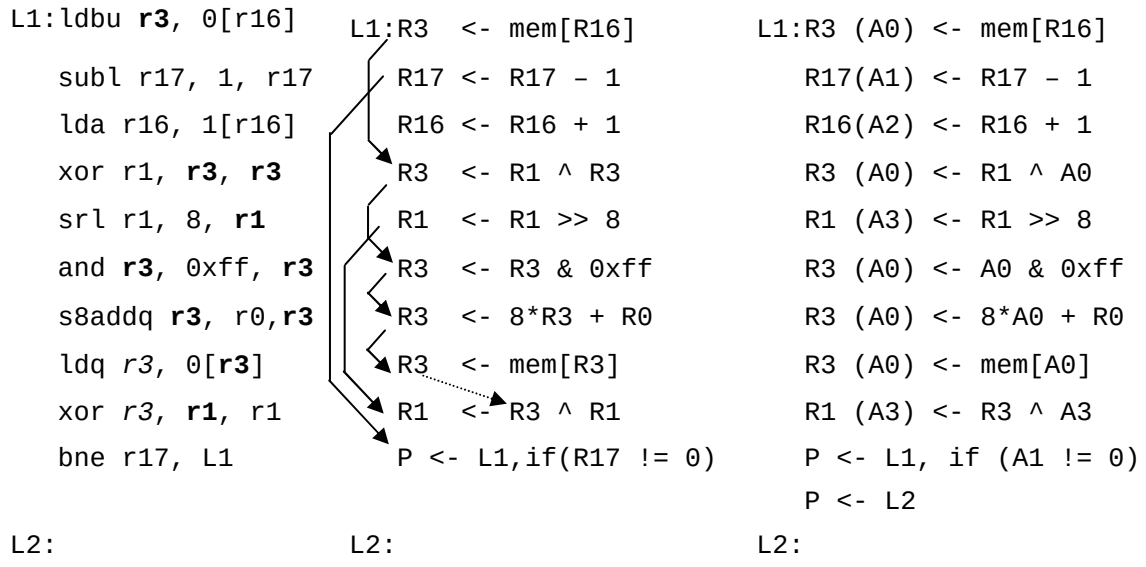
---

[9] One might get the impression that the strand assignment in the example is sub-optimal. For example, if the second `xor` instruction were assigned the accumulator 0, the long strand 0 (now 6 instructions) would seem to have a better chance of tolerating communication latency from the `srl` instruction to the second `xor` instruction. This particular assignment shown in the example came from the precise state maintenance requirement of the dynamic binary translation – the `R3` register of the second load is a live-out value that needs to be saved in a GPR anyway. Nonetheless, the parallel issuing of multiple strands works to hide this possible exposure of communication latency.

Note that the last conditional branch instruction is translated to a combination of a conditional branch and an unconditional branch for code cache chaining reasons; chaining is explained in Chapter 5 in more detail.

```
if (n) do {
    c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^ (c >> 8);
} while (--n);
```

(a) C source code




(b) Alpha assembly code[10] (c) Equivalent RTL notation (d) Translated ILDP ISA code

**Figure 2-4 Example code snippet from SPEC CPU2000 benchmark 164.gzip**

[10] For load instructions, the first register is the destination data register; for other operation type instructions, the result data is written to the right-most register.

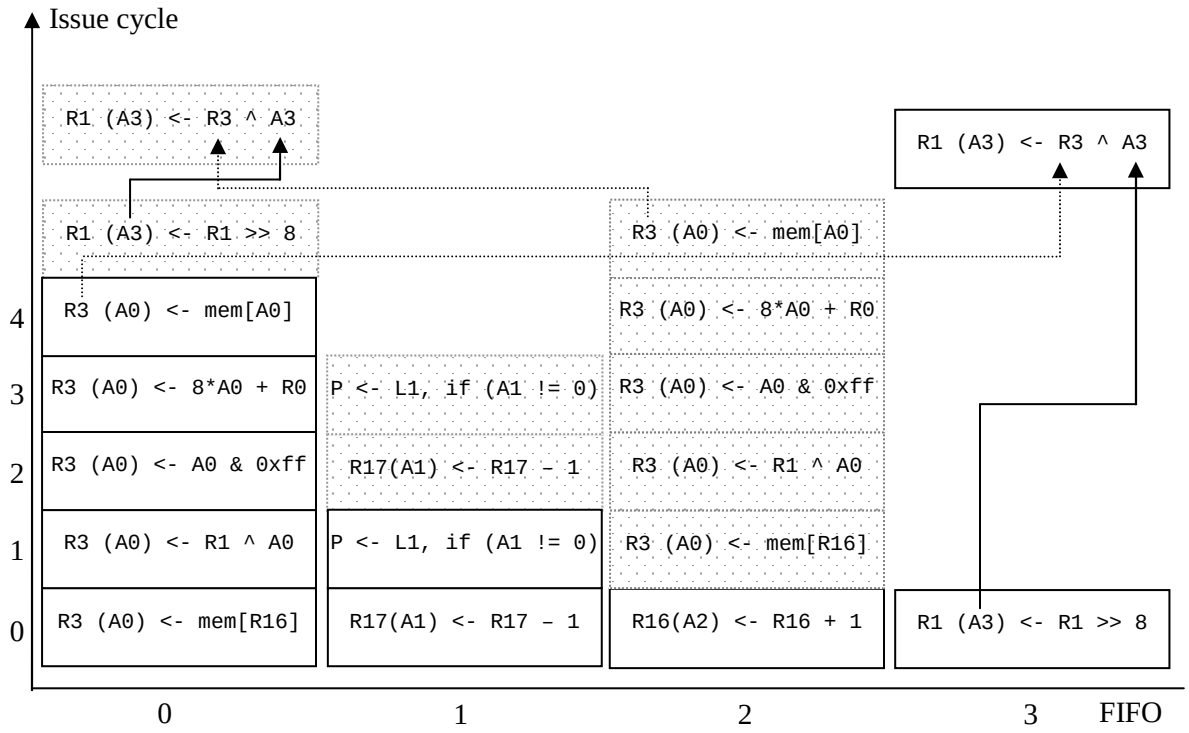



**Figure 2-5 Issue timing of the example code**

Figure 2-5 shows a possible issue timing of the translated instructions within FIFOs. Here four accumulators are used (A0 through A3). The ILDP instructions are steered to four different PEs (FIFOs). The instruction in the next loop iteration is shown in dotted boxes. It can be seen that the strands issue relatively independently, except where two strands converge to form inputs to the second `xor` instruction (denoted by the dotted arrows for GPR `R3` and the straight-line arrows for accumulator `A3`).

# 2.3 ILDP Instruction Set Formats

## 2.3.1 Summary of the Virtual ISA – Alpha EV6

Even among the RISC ISA families, the Alpha ISA is known for being a relatively minimalist design [214]. The ISA format was designed to be regular and orthogonal and hence, is simple to decode. All instructions are 32-bits wide. There are 32 integer and 32 floating-point registers, both 64-bits wide. Frequently executed instruction types such as memory and control transfer[11] instructions are given a range of op-code space and do not require secondary function field decoding. All instructions have a maximum of two input operands[12] and one or zero output operand. There is only one addressing mode: `register + offset`. There is no explicit condition code or special link register.

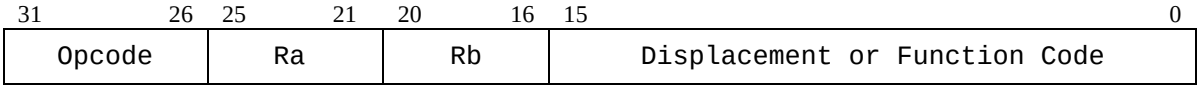
Conditional branches compare the input register operand value to zero. If necessary, a separate compare instruction is used for comparisons with a register values. Function calls can be made either with an unconditional branch (`BSR`) or an indirect jump (`JSR`). In fact, all indirect jumps have the same ISA semantics; save a return address, `PC + 4` to the output register and set `PC` to the input register value. They only differ in branch prediction hints – `JSR`, as well as `BSR`, gives the hardware a hint to push the return address onto a return stack (if the implementation has that feature) while `RET` is used as a hint to pop the return stack for the next fetch address. There is no conditional jump.

---

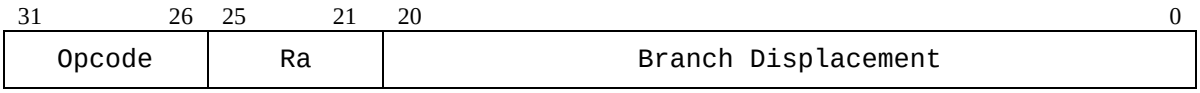
[11] Following the Alpha ISA convention the term **branch** is used for a control-transfer instruction whose target address is fixed. A register-indirect **jump** finds the target address by reading a specified register.

[12] Conditional move instructions can be considered to have three inputs in certain implementations.

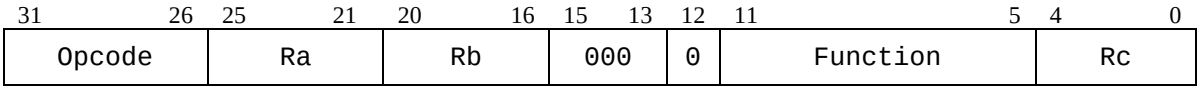
**Table 2-2 Alpha ISA instruction formats**



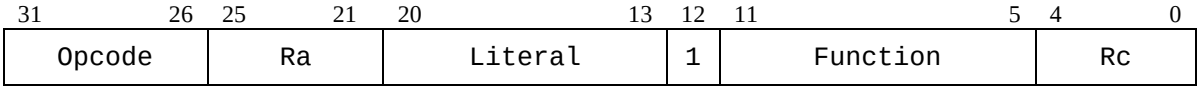

(a) Memory Instruction Format



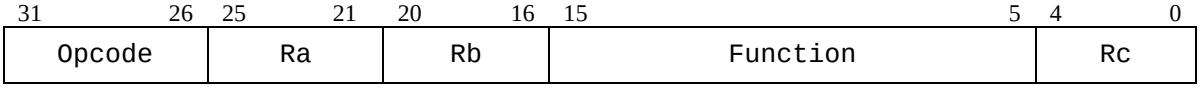

(b) Branch Instruction Format



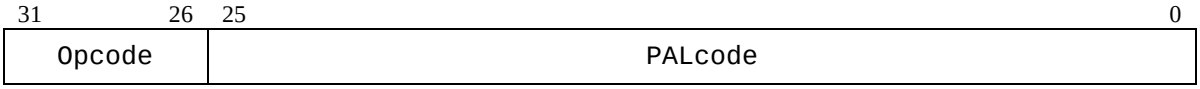

(c) Integer Operate-Register Instruction Format




(d) Integer Operate-Literal Instruction Format




(e) Floating-Point Operate Instruction Format




(f) PALcode Instruction Format

Being such a fine-grain ISA, Alpha architecture does not allow much freedom in implementing the ISA semantics. All registers have to be maintained as the architected state even though many of them are simply artifacts of compilation from high-level language program semantics.

### 2.3.2 Considerations for Dynamic Binary Translation

Designing an instruction set specifically for a hardware-software co-designed virtual machine system requires certain unique considerations. Due to the dynamic binary translation

nature of the paradigm, the range of optimizations allowed is often limited; this is made even harder when strict binary compatibility is required. On the other hand, co-design of special instructions and hardware support mechanisms (including special registers) allows a new degree of design freedom.

#### 2.3.2.1 Maintaining Precise Architected State

The precise trap recovery mechanism is a fundamental aspect of any co-designed VM system because it must provide exactly the same trap behavior as the V-ISA semantics define. Maintaining precise architected state is made relatively easy by the fact that the DBT system in the thesis does not reschedule instructions; hence values are produced in the same order as the original program. However, with an accumulator I-ISA this is not sufficient because some V-ISA GPR values are held in accumulators and may be overwritten prior to a trap. One possible solution is to *add copy-accumulator-to-GPR instructions* before instructions that overwrite an accumulator holding a value that will be live at a potential trap location. This is a fairly expensive solution, however, in terms of added instructions (as was shown in the early study [134]) and is not used in the thesis.

An alternative to adding explicit copy instructions as just suggested is to *embed GPR updates into the I-ISA format*. The resulting I-ISA format needs more bits to designate the result GPRs, so the number of instruction types that can be accommodated with 16-bit format is substantially reduced compared to the simple accumulator I-ISA format in the early study [133]. However, removal of copy-accumulator-to-GPR instructions, i.e., the decrease in the total number of instructions, works to offset the increase in the average instruction length.

In the finalized ILDP ISA format, every instruction producing a V-ISA result specifies a destination GPR to maintain architected state. Keeping the destination GPR identifier helps implement a simple precise trap recovery mechanism without affecting performance much. In the ILDP microarchitecture, a separate register file – off the critical path – is maintained solely to keep

the V-ISA GPR state for precise traps. Only the values needed for later computation, i.e., communication, are actually written to the "working" physical registers in PEs. This is possible because the I-ISA format distinguishes destination register types (further explained in section 2.3.3).

#### 2.3.2.2 Suppressing Code Expansion

Without special care, it is very easy for dynamically translated codes to have a larger effective memory footprint than the original code. The ILDP I-ISA in the research was designed to help suppress dynamic code size and instruction count expansion by using the following means:

- **Stay as close as the V-ISA format**: First and foremost, the ILDP I-ISA maintains the serial nature of the V-ISA semantics, unlike VLIW-based I-ISAs which inevitably introduce NOPs to fill the unused instruction slots. The ILDP I-ISA keeps one-to-one mapping for most instructions. However, there are some cases where a V-ISA instruction has to be converted into multiple I-ISA instructions. For example, when a source instruction has two global input register values, one of the inputs has to be converted into an accumulator by introducing an extra copy-GPR-to-accumulator instruction because the ILDP I-ISA instruction can only have one global input register specifier. Similarly, a conditional move instruction is decomposed into two ILDP I-ISA instructions.
- **Provide a small number of instruction sizes**: A variable-length instruction format complicates the instruction boundary detection somewhat, but the predecoding stage, which is employed anyway for identifying the possible control transfer points within an I-cache line, can be used for this purpose as in recent x86 implementations [226]. After experimenting with variety of alternatives, copy and NOP instructions along with four of the most frequently executed memory instructions without address calculation were assigned to 16-bit formats.

- **Allow dynamic micro-operation cracking by hardware**: In the early stage of the research, all memory instructions with non-zero address offset values were decomposed into an effective address calculation instruction and an actual memory access instruction [133][134]. Later it became clear that the benefits of bypassing address calculation for the single memory instruction were being offset by these extra effective address calculation instructions. Instead, most memory instructions with non-zero offset values stay as single instructions; they are kept as a single entity in most parts of the pipeline and are issued twice by the issue FIFO – a dynamic cracking mechanism. This is in line with the recent x86 implementations that try to keep internal instruction format at coarser-grain level as much as possible to reduce pressure on key pipeline buffers [63][90].
- **Compromise immediate and function field widths**: In general, an ILDP I-ISA instruction uses extra `mode` bits and 6-bit GPR identifiers so certain compromises in other bit-fields are unavoidable. The biggest goal being limited instruction count expansion, care was taken to maintain a one-to-one instruction mapping as much as possible. Except for the integer operate immediate formats, the immediate fields are reduced to make room for this extra information. For operate instructions, the function field and even the op-code encoding, to some extent, are re-organized for this purpose while the immediate literal width is kept 8-bit wide. If the immediate value in a memory or branch instruction exceeds the reduced size limit, the binary translator tries to decompose the V-ISA instruction into multiple I-ISA instructions first. If that is not possible, a trap-to-interpreter instruction is used as a final safety net.
- **Use specialized instructions or registers if necessary**: As stated earlier, the co-design paradigm combines specialized instructions and hardware support mechanisms. Most of the specialized instructions in this research are control transfer related and will be explained

further in the following section and Chapter 5. One important point is that there is a tension between the number of special instructions and the finite op-code space. Care was taken to limit the number of specialized instructions. On the other hand, the general purpose register identifier is expanded from 5 bits to 6 bits to accommodate the special registers only available to the virtual machine monitor software layer. Of the 32 additional register designators, only 8 registers are actually used; this limits the number of logical registers which determines the minimum number of physical registers.

### 2.3.3 ILDP Instruction Formats

The ILDP instruction formats are shown in the following sections. Note the use of `mode` bits; they are used to identify:

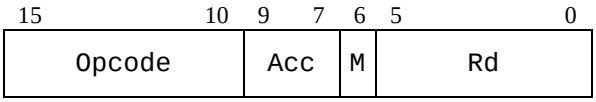
- If the instruction terminates a strand (end-of-strand)
- If the destination register value needs to be written to a working physical register (that is, if `Rd` needs to be renamed)
- Which register values should be used as inputs to the ALU; sometimes used with `extension` (that is, extended `mode`) bits

Head-of-strand information is implicitly encoded with either `mode` bits or the op-code itself. For example, a copy (from an accumulator to a GPR) instruction always starts but can never end a strand.

#### 2.3.3.1 Operate Instruction Formats

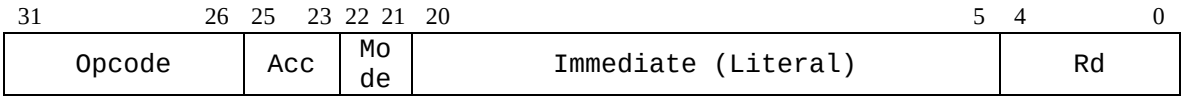
A challenge with the operate instruction format is that there are many different combinations of input operand value types – an accumulator, a GPR, and an immediate value. Another complication is that many operate type instructions such as subtractions are not commutative, i.e., the left-hand-side operand can not be swapped with the right-hand-side operand. To accommodate these requirements, the 4-bit `mode` field and 2-bit `extension` field (for non-immediate formats) are used. `GPR op Immediate` type operation complicates the instruction format design and hence, is not supported.

**Table 2-3 Operate instruction formats**



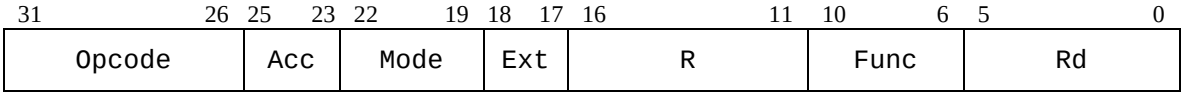

```
Mode[6]: floating-point?
Always start-of-strand
```

(a) Copy Format

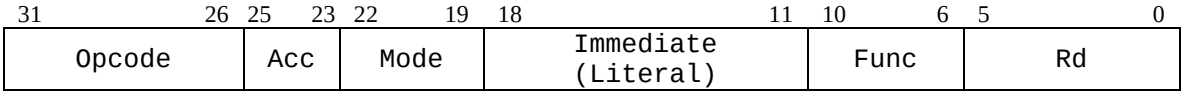



```
Mode[22]: End-of-strand?
Mode[21]: Save Rd?
If Mode[22:21] == 10, Rd += 32 (VMM register)
Cannot be start-of-strand
```

(b) Load Short Immediate Format (used for LDA/LDAH)




(c ) Integer/Floating-Point Operate Register Format



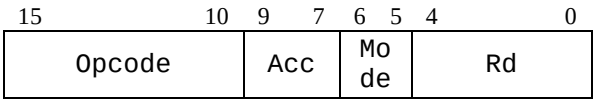

```
Mode[22]: End-of-strand?
Mode[21]: Save Rd?
Mode[20:19]          Ext[18:17]
00: R1  op R2        00: R1 == R2
    Start-of-strand  01: R2 == Rzero
                     10: R1 == Rzero
                     11: R1 == R2
01: Acc op R         11: invalid R, otherwise valid R
10: R   op Acc       11: invalid R, otherwise valid R
11: Acc op Imm       --: used as Immediate[18:17]
```

(d) Integer Operate Immediate Format

### 2.3.3.2 Memory Instruction Formats

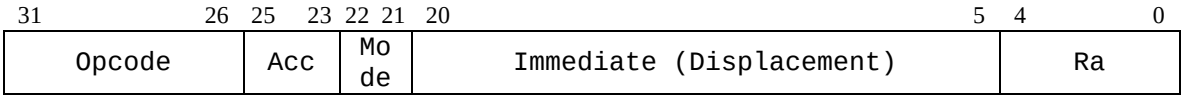
Ra and Rd represent the address and data registers, respectively. Note that depending the on Mode[19] value, a long memory instruction can either generate an effective address calculation instruction dynamically or not.

**Table 2-4 Memory instruction formats**



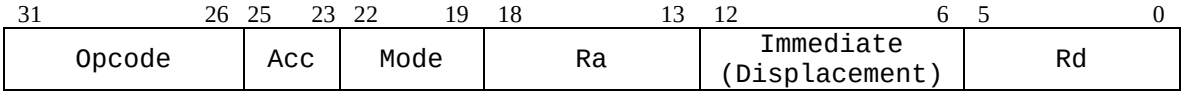

Mode[6]: End-of-strand?
Mode[5]: Load: Save Rd?, Store: Reserved
Cannot be start-of-strand

(a) Short Memory Format




Mode[22]: Use Ra?
Mode[21]: Check align?
Cannot be end-of-strand
Start-of-strand if Mode[22] == 1

(b) Effective Address Calculation Format (same as Load Short Immediate Format)

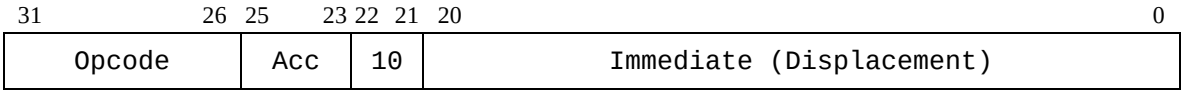



Mode[22]: End-of-strand?
Mode[21]: Load: Save Rd?, Store: Reserved
Mode[20]: Use Ra?, otherwise use Acc
Mode[19]: Use Offset?

(c) Long Memory Format

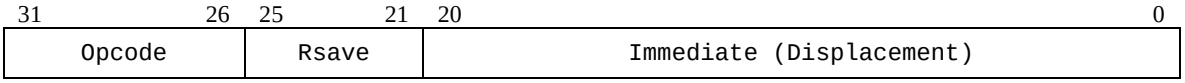
### 2.3.3.3 Control Transfer Instruction Formats

**Table 2-5 Control transfer instruction formats**



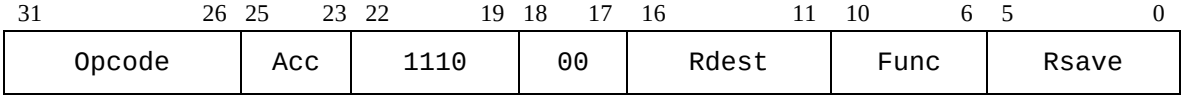

```
Cannot be start-of-strand
Always end-of-strand
```

(a) Conditional Branch Format



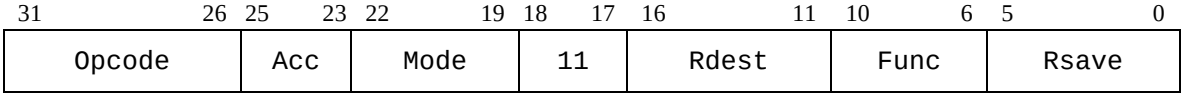

```
Always start-of-strand
Always end-of-strand
Rsave += 32 (VMM register)
```

(b) Unconditional Branch Format




```
Mode[22]: 1  (End-of-strand)
Mode[21]: 1  (Save PC+4 to Rsave)
Mode[20:19]: 10 (Use both Rdest and Acc)
```

(c) Conditional Jump Format (same as Register Operate Format)




```
Mode[22]: 1   (End-of-strand)
Mode[21]: 1   (Save Rsave)
Mode[20:19]
00: R1 op R2  (R1 == R2; Start-of-strand)
10: R  op Acc (invalid R)
otherwise: reserved
```

(d) Predicted Jump Format (same as Register Operate Format)

There are several things to note about control transfer instruction formats:

- The displacement field of a conditional branch contains the difference between the instruction's `PC + 4` and the target translation address, one-bit right-shifted.
- Unconditional branches are used to close a translation unit as in Figure 2-4(d). If the next translation exists at the time of translation, the displacement is set up properly according to the target address and `Rzero` (a constant zero register) is used for `Rsave`. If not, the difference between the instruction's `PC + 4` and the starting address of the dispatch table (a hash table that maps source binary program counter values to translated binary program counter values) lookup code is used for the displacement. When this place-holder instruction is executed, the instruction's `PC + 4` is saved to a VMM register (`Rsave + 32`) and control is transferred to the dispatch table lookup code. If the target is found by the table lookup, the saved address in the VMM register is used to identify the place-holder instruction that needs to be replaced with an unconditional branch instruction with a proper displacement value. This replacement action is often called a "patch" in dynamic optimizers and binary translators. Note that V-ISA unconditional branch instructions are either eliminated (for `BR` in the Alpha ISA) or converted to load-immediate instructions (for `BSR`) due to the code re-layout effect of the superblock-based translation. The slight change in the unconditional branch instruction semantics (save `PC + 4 to` a special VMM register) is intended to provide a fast, single-instruction mechanism for branching to the dispatch table lookup code.[13]

---

[13] HP Dynamo, which does not have the luxury of specialized instruction support, uses a four instruction "stub" code [14] for each branch whose target was not found at translation time. A place-holder

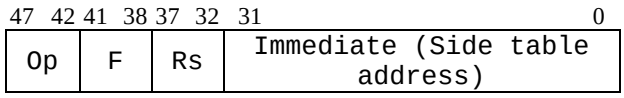
- A conditional jump is generated as a place-holder for a V-ISA conditional branch whose branch target was not found in the translation cache at the time of the translation. By convention, `Rdest` is set to a special VMM register that contains the starting address of the dispatch table lookup code. The instruction's `PC + 4` is saved to `Rsave`; If the translated target address is found by the lookup, the `Rsave` value is used to identify the conditional jump for patching. The `Func` field contains an abbreviated form of the original V-ISA instruction's op-code and is used to generate the `Opcode` field of the replacement conditional branch instruction for the place-holder conditional jump. As with the slight change in the unconditional branch semantics, the introduction of a conditional jump instruction is intended to provide a fast, single-instruction mechanism for branching to the dispatch table lookup code.
- The semantics of the ILDP predicted jump instruction are slightly different from the conventional definition – a conventional jump always jumps to the target. Here, if the jump target address prediction is not correct, the next sequential instruction is executed. The DBT system always generates a backup unconditional branch to a separate dispatch lookup code for indirect jumps, right after a predicted jump instruction. The predicted jump instruction, and related hardware support mechanisms are further discussed in section 5.2.

---

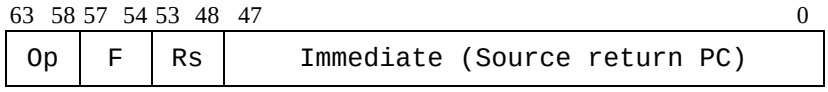
instruction branches to the stub code; control is finally transferred to the dispatch lookup code using the stub as a springboard.

### 2.3.3.4 Load Long Immediate Formats

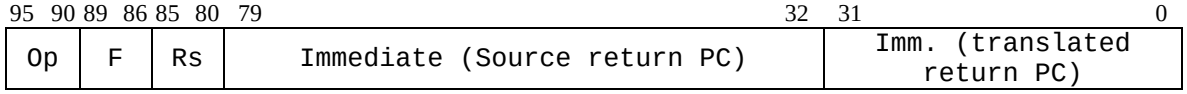
**Table 2-6 Load long immediate formats**




(a) Load Address Instruction




(b) Save Return Address Instruction




(c) Push Dual-address Return Stack Instruction

A load long immediate instruction can have three different combinations of a 48-bit Alpha V-ISA address and a 32-bit ILDP I-ISA address. The first type, a load address instruction, is used to keep track of a side table of potentially excepting instructions (PEIs) associated with the translation. Trap recovery using side tables is described in section 4.2.1. The remaining two types are used for translating a function call instruction. A source function call instruction is decomposed into two parts: (1) to save the return address, (2) to branch or jump to the target address. Load long immediate instructions are used to save the V-ISA return address (`PC + 4` of a V-ISA function call instruction) into a register. Pushing a return stack using these two instruction types is described in section 5.2.3.

It is true that this operation can be implemented using a sequence of multiple load short immediate instructions (equivalent to `LDA/LDAH` in the Alpha ISA) but a typical general-purpose load (short) immediate instruction does not contain a hint to push the return stack.

## 2.4 Related Work

### 2.4.1 Execution Paradigms Based on Register Hierarchy

Franklin and Sohi studied register value usage locality in [86]. Multiscalar [220] and Trace processors [197][233] exploit this locality by separating the live-in and live-out values (assigned to global registers) from the locally dead values (assigned to local registers contained in distributed clusters) inside a task or trace (an collection of multiple basic blocks). Although the research here shares some objectives (such as reducing the complexity of key pipeline subsystems) with those previous studies, there are many important differences. Most importantly, both Multiscalar and Trace Processors pursue the parallelism between tasks/traces that span multiple basic blocks. As such, global and local values are separated based on their liveness within those task/trace boundaries. In contrast, the instruction set presented here distinguishes register values based on their *degree of use* as well as their *lifetimes*. This leads to the identification of strands (of several instructions) that can run in parallel. In general, this thesis research does not pursue ILP as aggressively as in those paradigms. Therefore, hardware-intensive and complex-to-verify speculation techniques are not used in the thesis, other than the most essential ones such as branch prediction.

There has been plethora of proposals on program execution models based on the idea of dependence-based instruction steering for cluster microarchitectures. Palacharla, Jouppi, and Smith's work [175] suggested *dispatch-stage* instruction steering to multiple issue FIFO buffers to

reduce issue window complexity. Although the thesis shares dependence-based dispatch-stage steering idea with their work, the objective here is to reduce the complexity of *all* key subsystems in the pipeline by starting afresh with a new (but not radically different) instruction format specifically designed to fit modern technology constraints. In the Multicluster model [79], the global registers that are replicated in the distributed clusters are statically selected based on a heuristic. The stack and global pointers in the compiler calling convention belong to the global registers. All other registers, i.e., local registers are statically partitioned (e.g., the even-numbered registers in a cluster and the odd-numbered in another). If not all registers of an instruction are in the same cluster, a master copy (for actual computation) and a slave copy (for communicating a required register value to the master cluster) of the instruction are dispatched to each cluster. It is the static nature of their register partitioning that practically mandates a specialized register allocation algorithm in a compiler. The PEWS model [130] can be thought as a data-dependence-driven version of the Multiscalar paradigm, combined with a trace cache. A trace pre-processing unit identifies intra-trace register dependences. Live-in and live-out registers are assigned a register queue (containing multiple versions of an architected register) entry; locally dead values are not assigned an entry. However, the register queue has to provide enough versioning entries for *all* architected registers the ISA defines – a newly introduced complexity-critical logic. Complex memory versioning techniques are also used to pursue aggressive ILP.

### 2.4.2 Related Instruction Set Ideas

Lozano and Gao [150] tried to reduce register pressure by allocating short-lived register values to architecturally visible reservation stations. Recent studies by Butts and Sohi observed similar program characteristics as in this chapter and suggested microarchitectural techniques to dynamically eliminate “useless” instructions [36] and to predict value communication patterns [37].

In the work of Martin, Roth, and Fischer [153], the end of a register value's lifetime is explicitly encoded in instructions. In contrast, the ISA presented here encodes the end of the *dependence's lifetime* in instructions.

The instruction set in the research is very much inspired by the S. Cray scalar ISAs (just as the 1980s microprocessor RISCs were). However, in a sense, the accumulator-oriented instruction set in the research follows the Cray ISAs more closely than the microprocessor-based RISCs. In particular, the ISA in the thesis uses hierarchical register files with a very small number of registers at the top of the hierarchy, variable length instructions, and in-order instruction issue (albeit within individual processing elements). Even though the technology was quite different when Cray's designs were undertaken, the issues of interconnect delays, power consumption, and design complexity were of critical importance, just as they are today, and will continue to be in the future. In effect, the accumulator-oriented ILDP ISA is a cross product of two Cray-2 designs. One is an abandoned Cray-2 design [53] that had a single renamed accumulator and a general register file of 512 elements. The completed Cray-2 design [54] had 8 integer registers, 64 lower level registers, and used conventional three-operand instructions.

Other ISAs that explicitly model inter-instruction communications, such as the Grid Processor Architecture [171] or RAW [146][234], can be considered as different ways of implementing ILDP. Those systems are primarily designed to maximize microarchitecture scalability for high ILP applications such as multi-media processing. As such, their ISA semantics are radically different from conventional ISAs. To facilitate efficient dynamic binary translation, the ILDP instruction set in this research is rather evolutionary in comparison.

# Chapter 3 ILDP Microarchitecture

The strand per accumulator concept of the ILDP ISA is reflected in the moderately distributed ILDP microarchitecture. The microarchitecture consists of a pipelined front-end of modest width that feed a number of distributed processing elements (PEs). Each PE contains an instruction issue FIFO, a local accumulator and local copy of register file, and other functional units. Taken collectively, the multiple sequential PEs implement a form of multiple-issue out-of-order instruction processing.

This chapter starts with the overview of the ILDP microarchitecture. A unique combination of a physical register file based register renaming model and a capture-before-issue operand capture model is described, and then compared with the conventional register renaming and operand capture models used in typical superscalar designs. This is followed by detailed descriptions of key pipeline subsystems and discussion of several L1 D-cache organization options. Related work on existing complexity-effective processor designs and research proposals complete the chapter.

## 3.1 ILDP Microarchitecture Overview

### 3.1.1 Overall ILDP Pipeline

Figure 3-1 shows a high-level block diagram of the ILDP pipeline. The boxes represent pipeline stages. The arrows represent instructions flowing through the pipeline; think of these lines as pipeline “lanes”. The thick vertical lines represent the pipeline buffers where instructions can be stalled waiting for the resources to be available. Various replay and cache miss paths are shown with dotted lines.

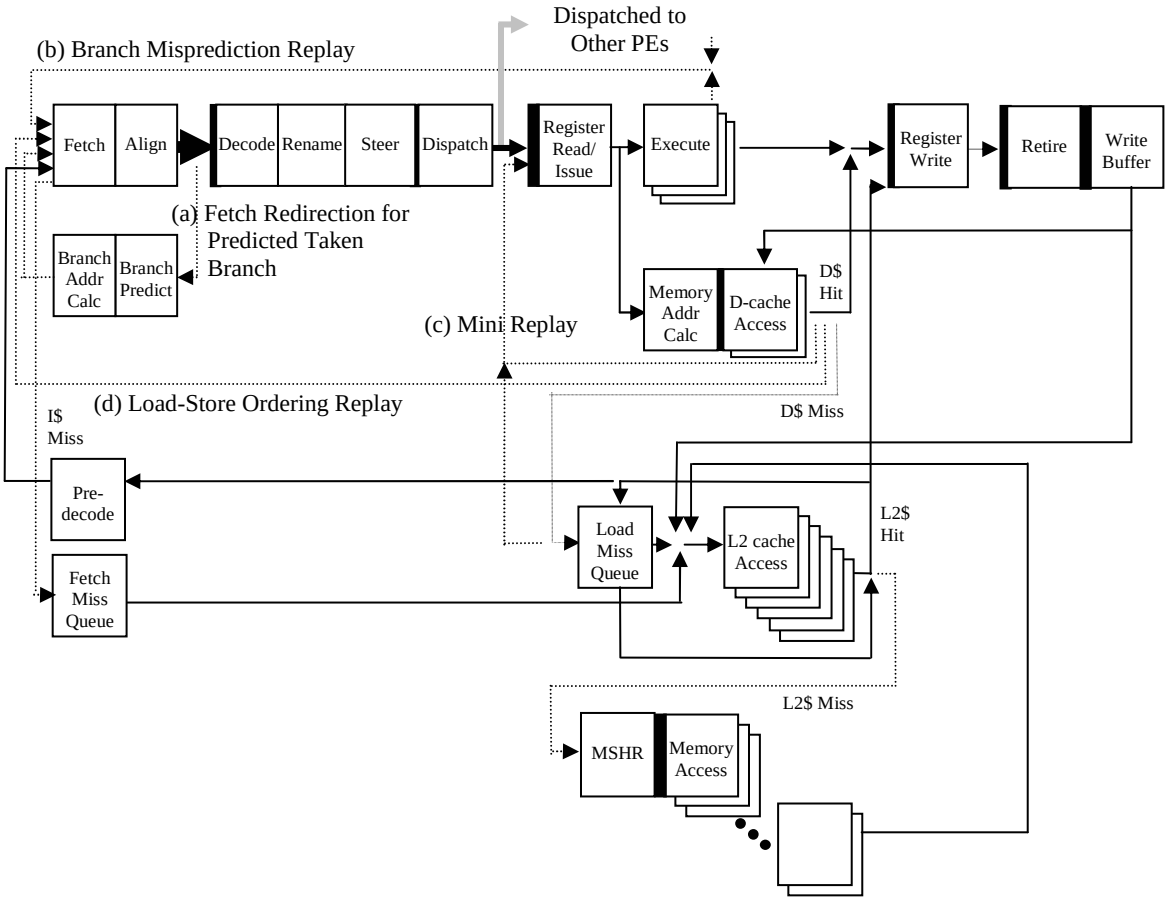



**Figure 3-1 High-level block diagram of the ILDP pipeline**

On an I-cache miss, translated ILDP instructions are fetched from the translation cache in hidden memory to the I-cache. The average number of instructions fetched in a cycle will be higher than with conventional processors due to the code re-layout that is a byproduct of superblock-based binary translation (superblocks and associated code re-layout effects are described in section 4.1.2). Predecoded instruction boundaries are used to break instruction words into individual instructions in the *align* stage. Instruction formats are recognized using the `opcode` field in the *decode* stage and used to further extract the variable-width `mode` field. The *renaming* stage renames only GPRs. The

accumulators are not renamed in this stage; they undergo a simpler type of renaming as a byproduct of steering to the sequential PEs.

The *steering* logic maps strands of GPR-renamed instructions to one of the FIFO issue buffers, depending on the accumulator to be used. Any instruction that has an accumulator as an output, but not as an input, is steered to the first empty FIFO; consequently, the logical accumulator is renamed to the physical accumulator. Any later instruction that uses the same accumulator as an input is steered to the same FIFO. Whenever all the FIFOs are non-empty and a new accumulator is called for, the steering process uses a heuristic to choose a FIFO to use next. Once their accumulator numbers are converted to corresponding FIFO numbers, instructions are *dispatched* to their distributed back-end FIFOs. Each FIFO feeds a sequential processing element with its own internal physical accumulator.

The instructions in a FIFO form a dependence chain, and therefore issue and execute sequentially. When the GPR value of the FIFO-head instruction is available, the instruction *issues* and begins execution. The functional units can be single or multi-cycle, but in general do not require pipelining. Because accumulator values stay within the same PE, they can be bypassed without additional delay. However, GPRs are kept coherent and values produced in one PE must be communicated to the others. This will take additional clock cycles. The network for communicating GPR values can be a bus, a ring, or point-to-point. Multiple bus interconnects are used in the evaluation. The bandwidth requirements are modest and, as will be shown later, performance is relatively insensitive to this latency.

L1 D-cache can be organized in several different ways based on the required degree of distribution. For highly clustered configurations, L1 D-cache is replicated across the PEs. Less aggressive configurations can choose to share cache read ports among multiple FIFOs. Combined

store data values from the write buffer (WB), as well as line fill data from the L2 cache, are broadcast to the replicated D-caches to keep their contents coherent.

Memory data can also be forwarded from the store queue (STQ) to load instructions. For fast lookup, the STQ is replicated for each L1 D-cache read port. Every load instruction consults the STQ *after issue* for possible forwarding or conflicts with preceding stores. STQ entries as well as other ordering enforcement buffer entries are allocated prior to the out-of-order dispatch stage and use separate paths from the main dispatch lanes. Every issued store instruction checks the load queue (LDQ) for possible conflicts with younger load instructions that were issued before the store instruction. The LDQ is relatively insensitive to the latency and is shared by the PEs. A simple PC-base memory dependence predictor is used to limit the number of costly memory ordering replays.

Regarding the L2 cache and the bus interface, there is no fundamental difference between the ILDP pipeline and conventional out-of-order superscalar pipelines. Multiple requests arbitrate for the unified, single-ported, write-back, write-allocate L2 cache. Normally, the load miss queue (LMQ) is given the highest priority because most often it is the missed loads that are on a program's critical path. The fetch miss queue (FMQ) is assigned the second highest priority. A combining write buffer[15] holds retired stores to the write-through, no-write-allocate L1 D-cache until a preset "watermark" threshold is reached. To avoid long starvation, WB watermarks and watchdog timeouts are given priority over an FMQ timeout, which in return is given a higher priority than LMQ requests. The arbiter runs at the L2 cache's operating frequency, usually a fraction of the core pipeline's frequency.

---

[15] Use of a combining WB makes the memory ordering model of the ILDP pipeline the same as or weaker than Processor Consistency model [108]. A snooping policy on the LDQ will further determine the ordering model.

A missed request for the L2 cache is assigned a combining miss status history register (MSHR) entry, as well as a possible replacement copy-back request. The memory bus interface runs at the fraction of the L2 cache frequency.

### 3.1.2 Speculative Instruction Execution and Replays

Modern high-performance processors invariably contain a certain number of speculative execution mechanisms. This is mostly for achieving higher performance, but some speculation mechanisms also help ease the design of the pipeline, especially the data cache access mechanisms [242]. The ILDP pipeline contains a number of speculation loops. Some of the more important speculation loops in Figure 3-1 are explained below.

**(a) Fetch redirection for predicted taken branches**: Even if a control transfer instruction is correctly predicted, it still takes multiple cycles to redirect fetch. Combined with fetch-width-unaligned addresses of branch instructions and their targets, this reduces the effective fetch bandwidth[16].

**(b) Branch misprediction replay**: The ILDP pipeline redirects fetch as soon as a mispredicted branch or jump is found in the execute stage. Then, register renaming is blocked until the

[16] Some of the techniques used by modern high performance processor designs with non-trivial pipeline depth to reduce the number of fetch bubble cycles include: (1) predecode I-cache lines with control transfer and instruction boundary information, (2) limit I-cache associativity and size to keep its pipelining depth to a moderate level, (3) use fast but inevitably less accurate next fetch address predictors such as a line predictor [131]. In general, the impact of the fetch bubble cycles is more critical for variable-length ISA processors [64][98].

mispredicted instruction retires and hence, the machine state is up-to-date.[17] The distributed nature of the ILDP pipeline leads to out-of-order resolution of control transfer decisions.

**(c) Mini replay**: A mini replay causes previously issued instructions to reappear in the issue buffer and issue again after a preset number of cycles, but do not flush the entire pipeline. Although mini replays are less costly compared with full replays, they still affect performance and more importantly, waste energy. The ILDP pipeline can generate a mini replay in the following scenarios: (1) when a load instruction hits an older, address-matching store instruction in the STQ but the store instruction's data is not yet available, (2) when a load address conflicts with a L1 D-cache line fill operation, (3) when the LMQ cannot accept an incoming missed load instruction due to conflict or capacity limit.

**(d) Load-store ordering replay**: In the ILDP microarchitecture in the thesis, load instructions are allowed to issue speculatively without waiting for all older store addresses to be resolved. In other words, memory dependences are speculated while register dependences are strictly enforced. If it is found later that a younger load instruction was issued before a matching older store instruction, all instructions including the offending load instruction are flushed from the pipeline and fetch is redirected to the offending load instruction's address. Both LDQ and STQ can generate this type of replay: (1) when a load finds an overlapping store in the STQ that cannot forward data due to misalignment of addresses or data size, (2)

---

[17] Some older designs take snapshot of the machine state at certain boundaries, e.g., conditional branches for faster recovery [241]. This technique is less popular for current-generation designs with non-trivial pipeline depth due the following reasons: (1) as the machine depth increases, the required snapshot size also increases [7], (2) the time between branch resolution and retirement can be overlapped with redirection time up to the register rename stage.

when a store finds a matching, younger load in the LDQ that has already issued. To limit the number of these costly replays, a simple dependence predictor that records PCs of the offending loads is used [131][165]. This predictor entry is cleared after a preset number of cycles.

Note that prefetches (load instructions into a constant zero register) are properly recognized; when a prefetch leads to a replay condition, it is simply marked complete and replay is not performed.

### 3.1.3 ILDP Operand Capture Model: A Unique Middle Ground

Register renaming is a popular microarchitecture technique that is used to remove false dependences between instructions, thereby allowing parallel execution of those instructions. There are two common methods for register renaming used in typical superscalar designs [215]. In the first, a physical register file, typically larger than the logical register file, is used. A rename table is used to map a physical register with the current instance of a logical register. Typically, a physical register that was allocated for an instruction's result value is freed when the next instruction that writes to the same logical register is retired from the reorder buffer.

The second model uses the reorder buffer (ROB) for renaming. In this model, typically a register file that is the same size as the logical register file is used to keep the latest committed register values. In addition, when an instruction completes execution, its result value is written to its reorder buffer entry. Here a register value can come from the register file (if the value was retired), a ROB entry (if the value was computed but the producer instruction is still in-flight), or the bypass network. A future file [211] style renaming mechanism used in AMD Athlon [64] and Opteron [129] processors can be regarded as a form of reorder buffer based renaming model in a broad sense. Here when an instruction is dispatched, it first checks the future file. If the value is not available at

the time, the instruction is given a tag of the producer instruction instead. Although instructions do not check the ROB per se, they do monitor the tags on the result bypass network constantly and receive the in-flight values from the bypass network.

Theoretically, these two register renaming models can be combined with any of the two register operand value capture timings: before or after the instruction issue. In reality, designs that employ the physical register model tend to capture operand values after instruction issue. On the other hand, designs that use the reorder buffer for renaming tend to capture values before instruction issue. Figure 3-2 shows a classification of register operand capture models.

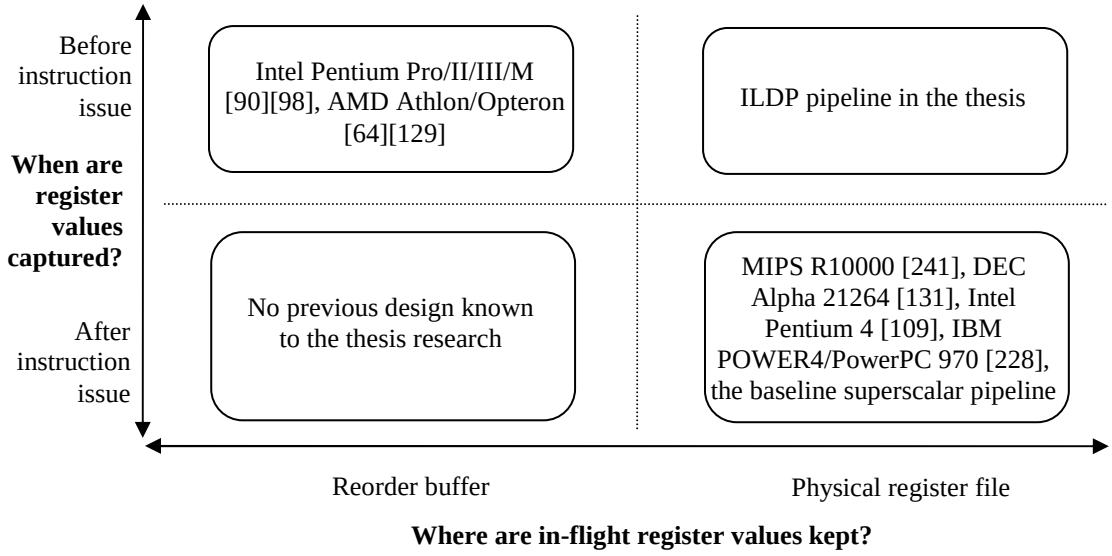
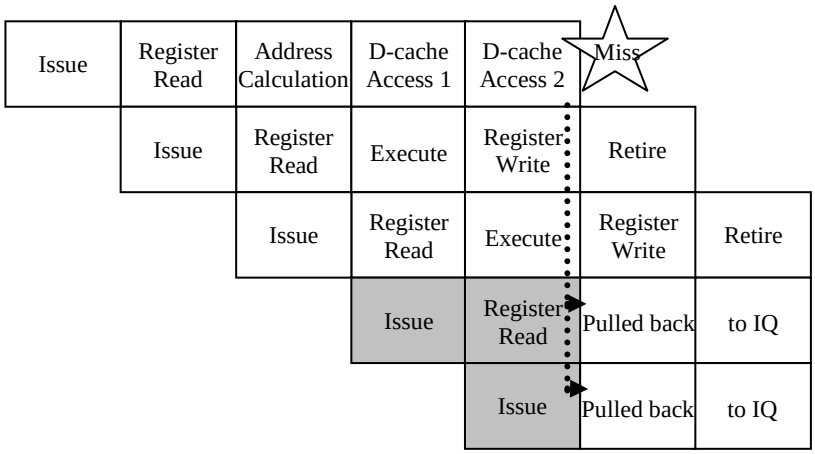
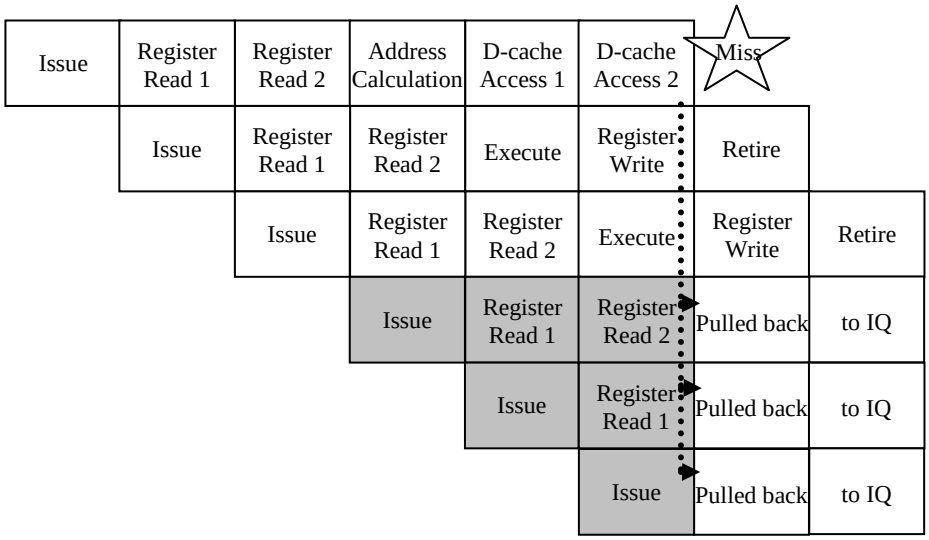



**Figure 3-2 Spectrum of register operand capture models**

There are good reasons behind these tendencies. For the designs that capture register values after issue, having a single source of operand values (e.g., a physical register file with complete internal bypasses) in the paths to the functional units makes the physical design of the issue lanes easier.




(a) Single-cycle register read: two-cycle shadow




(b) Two-cycle register read: three-cycle shadow

**Figure 3-3 Shadow cycles in load latency speculation**

A problem with this model is that often the instructions that are dependent on a load instruction are issued before confirming the availability of their operands. If the load misses in the L1 D-cache, some instructions in the "shadow" cycles, shown in Figure 3-3, capture wrong data. These instructions need to be pulled back and mini-replayed later. It is not impossible to design a

physical register file based machine that issues instructions non-speculatively even for those dependent on loads but doing so effectively increases the load-to-use latency.

The designs that capture register values before issue – sometimes called data capture machines [31] – do not suffer this type of mini replay. In return, this type of machine needs to look for data in multiple places; first the designated register storage (one time lookup), then the bypass network (continuous monitoring). Therefore it doesn't make good sense to use the fairly big physical register file before dispatching instructions to the reservation stations. The content-addressable-memory (CAM) style reorder buffer lookup was justifiable in some older designs if the ROB size was reasonably small [98]; typically the ROB is physically located between the register renamer and the reservation stations. However, the associative nature of CAM lookup puts a scalability limit on the ROB in this model. On the other hand, a future file is smaller and faster than a ROB but only a few percentages of the instructions can pick up their values from the future file. Therefore the design tends to rely on the bypass network heavily [64]. This leads to a very tight integration of the reservation stations and the bypass network. For example, if instruction dispatch takes two cycles, the reservation station needs extra wires to cover the newly introduced intermediate stage.

Table 3-1summarizes the two models as well as the unique ILDP model where instructions capture register operand values from the physical register file (or the accumulator) before issuing from a FIFO. From the table, it can be seen that the ILDP operand capture model combines the best of both models. The ILDP model does not require load latency speculation and a large fan-out bypass network.

**Table 3-1 Comparison of register renaming/operand capture models**

| | Physical register/capture after issue model | Reorder buffer/capture before issue model | ILDP model (physical register/capture before issue model) |
|---|---|---|---|
| Renaming | Physical register | Reorder buffer | Physical register, steering |
| Operand value sources | Physical register file | Architected register, reorder buffer, bypass network (or future file, bypass network) | Physical register file, accumulator |
| Operand lookup | RAM | RAM, CAM | RAM |
| Operand capture timing | After issue | Before issue | Before issue |
| Load latency speculation | Yes | No | No |
| Advantage | Physical register file provides a single unified source of register values, RAM access | Lack of shadow cycles allows simpler mini-replay implementation and reduction of total mini replays | No shadow cycle, simplified operand capture |
| Disadvantage | Shadow cycle complicates mini-replay mechanism; the number of shadow cycles increases with the physical register file access cycles | Multiple places for operand lookup complicate physical design; CAM lookup puts a scalability limit | Both physical register file and architected register file exist |

## 3.2 ILDP Pipeline Subsystems

### 3.2.1 Front-end Pipeline

#### 3.2.1.1 Instruction Fetch and Branch Prediction

As will be shown in Chapter 5, basic blocks are automatically re-laid out in the most frequently executed order during dynamic binary translation. As a result, the number of taken branches is reduced. For this type of fetch stream, a wider fetch mechanism coupled with a (potentially slower) multiple-prediction branch predictor works better than a combination of a narrower fetch and a fast predictor. This relaxes the timing requirements of the fetch subsystem (loop (a) in Figure 3-1). There are small differences, mostly from the differences in the control transfer instruction semantics. For example, even when a return stack is popped for a return instruction, a branch (direction) predictor is also probed because a return is now conditional in the translated ILDP code. More details on the control transfer mechanisms are provided in section 5.2.

Multiple instruction sizes in the ILDP ISA do complicate the align stage. Possible instruction boundaries are now every two bytes, in contrast to every four bytes in traditional RISC ISAs. To help reduce the complexity of the align stage, instruction boundaries are detected when the instructions are predecoded for branch hints before being put into the I-cache [226]. The load long immediate instructions that are larger than four bytes can only be aligned as the first instruction in a given cycle. This limit helps to reduce the complexity of multiplexed alignment network.

#### 3.2.1.2 Instruction Decoding and GPR Renaming

Once instructions are properly aligned in the fetch buffer, the rest of the front-end pipeline up to the renaming stage remains largely the same as conventional superscalar designs. One

possible complication (compared to the simple Alpha ISA format) is the variable length `mode` bit field (plus the extended mode bits for integer operate type instructions). These bit fields carry lower-level information such as end-of-strand information and the types of the input register operands. The ILDP ISA format was designed in such a way that this sub-decoding can be performed at the same time when the GPRs are renamed.

The register rename bandwidth is reduced because the ILDP ISA uses only one input and one output GPR. Two read ports (one for the input GPR, another for the old mapping of the output GPR) and one write port (for the output GPR) to the rename table are assigned to each rename lane. For a 4-way front-end, the ILDP pipeline requires total of 8 read ports and 4 write ports while a comparable superscalar pipeline requires 12 read ports (two input GPRs per instruction) and 4 write ports.

#### 3.2.1.3 Instruction Ordering Setup and Accumulator Renaming

Once the GPR dependences are set up by the physical register numbers, a reorder buffer entry is assigned to each renamed instruction as in conventional out-of-order processors. Other ordering maintenance buffers such as the register scoreboards and the store queue, however, are treated somewhat differently because they are located in the pipeline back-end. Furthermore, these replicated buffer contents must be kept coherent. Note that the instruction dispatch process (explained in the next section) is a form of out-of-order processing; a logically younger instruction can be dispatched to its FIFO before an older instruction can be dispatched to a different FIFO. For this reason, ordering information needs to be sent to the back-end while instructions are still kept in order (*before* the dispatch stage). This requires separate paths from the dispatch lanes.

Although mapping of the accumulator of an instruction to a physical accumulator can be performed as soon as the accumulator identifier is read, the actual renaming is done in the last in-order pipe stage before the dispatch stage. This is to keep the FIFO occupancy information (sent

from the back-end pipeline) as current as possible. As a result, accumulator renaming is done in the same cycle as the ordering setup process.

Instructions access the accumulator rename table in much the same way as the GPR renaming. Unlike GPR renaming, however, old mapping information is not needed because the end-of-strand information is explicitly marked within the instruction. Therefore an ILDP instruction needs only one read port and a write port to the accumulator rename table.

For those instructions that are start-of-strand (identified in the decode process), a new FIFO number should be assigned. A simple heuristic is used; if there is a free FIFO available, the accumulator is mapped to the FIFO. If not, the first FIFO with a dead strand that is not stalled by a missed load is assigned. If no such FIFO is available, accumulator renaming is stalled. A potentially better heuristic would be to give priority to the unblocked FIFO among all FIFOs with a dead strand but that would increase the complexity of the accumulator renamer. Coupled with the end-of-strand update (freeing a FIFO), the multiple FIFO assignments form a priority encoder chain between the accumulator-renamed instructions. This is potentially one of the most complexity critical parts of the pipeline. However, the number of instructions under consideration is fairly small (four in the evaluation) compared to the typical size (about 8 to 12) of the select logic, a key priority encoder chain in the conventional out-of-order issue logic.

#### 3.2.1.4 Instruction Dispatch

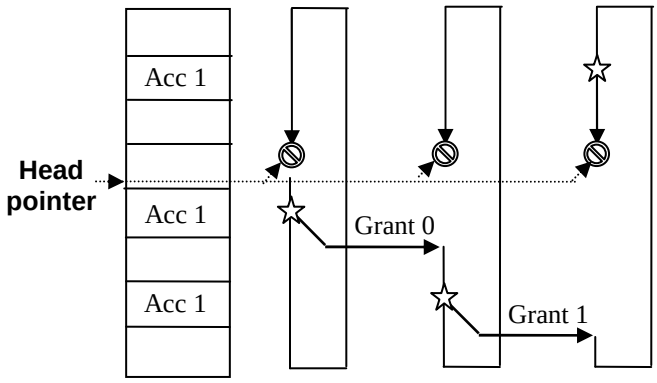
As with the conventional out-of-order superscalar processors, the selectors (sometimes called pickers or functional unit ports) representing the execution resources “select” the matching instructions. In the ILDP microarchitecture, a selector represents an issue FIFO. A key difference is that the ILDP dispatch logic does not involve the “wakeup” phase in the conventional out-of-order issue logic [175] because the dispatch logic in the ILDP pipeline works only on the static FIFO numbers, and hence does not use the dynamically updated “ready” bits. In a sense, the select phase

of the traditional out-of-order issue logic is performed separately by the dispatch stage in the ILDP pipeline while the wakeup phase is handled by a simple combination of the replicated physical register file and the accumulator in a PE. Table 3-2 summarizes the differences between the ILDP dispatch logic and the traditional out-of-order issue logic.

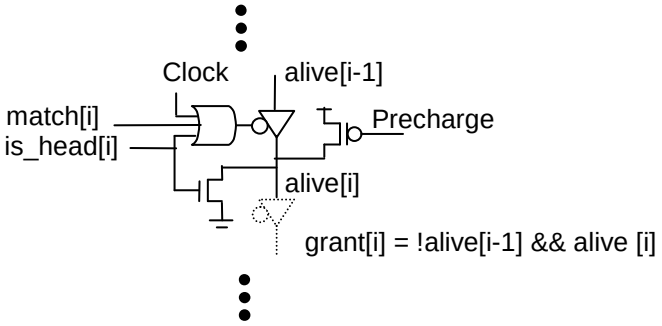
**Table 3-2 Comparison of ILDP dispatch logic and out-of-order issue logic**

| | Out-of-order issue logic | ILDP dispatch logic |
|---|---|---|
| Selector represents | Functional unit | Issue FIFO |
| Selection criteria | Select matching *ready* instructions | Select instructions with the matching FIFO number |
| Function | Select plus wakeup | Only select |
| Bandwidth | functional unit bandwidth | Dispatch lane width |

Note that in a typical out-of-order issue logic, the bandwidth of the selection logic is fairly low, typically one for each functional unit. On the other hand, a FIFO in the ILDP pipeline can accept as many instructions as the dispatch lane width allows. For this type of selector, a logic style that scales well with multiple selections is preferred. Here circuit speed is less important due to the omission of the wakeup phase. A good fit is multiple ring-connected “propagate” style selectors [106]. Figure 3-4 shows the basic idea and a sample implementation using pre-charged tri-state buffers. Note that the total latency of the selector does not increase with the number of entries that can be selected in a cycle.




(a) concept




(b) a sample implementation using tri-state buffers

**Figure 3-4 Scalable dispatch logic for the ILDP pipeline**

### 3.2.2 Processing Element

Figure 3-5 shows the internals of a PE with a copy of the fully replicated L1 D-cache. As dependent instructions issue in order, the circuit design complexity is maintained at the level of a single-issue, in-order pipeline. Note the simplicity of the internal single-cycle bypass. The replicated GPR has only one read port. The number of write ports is determined by the desired bandwidth of the global communication network.

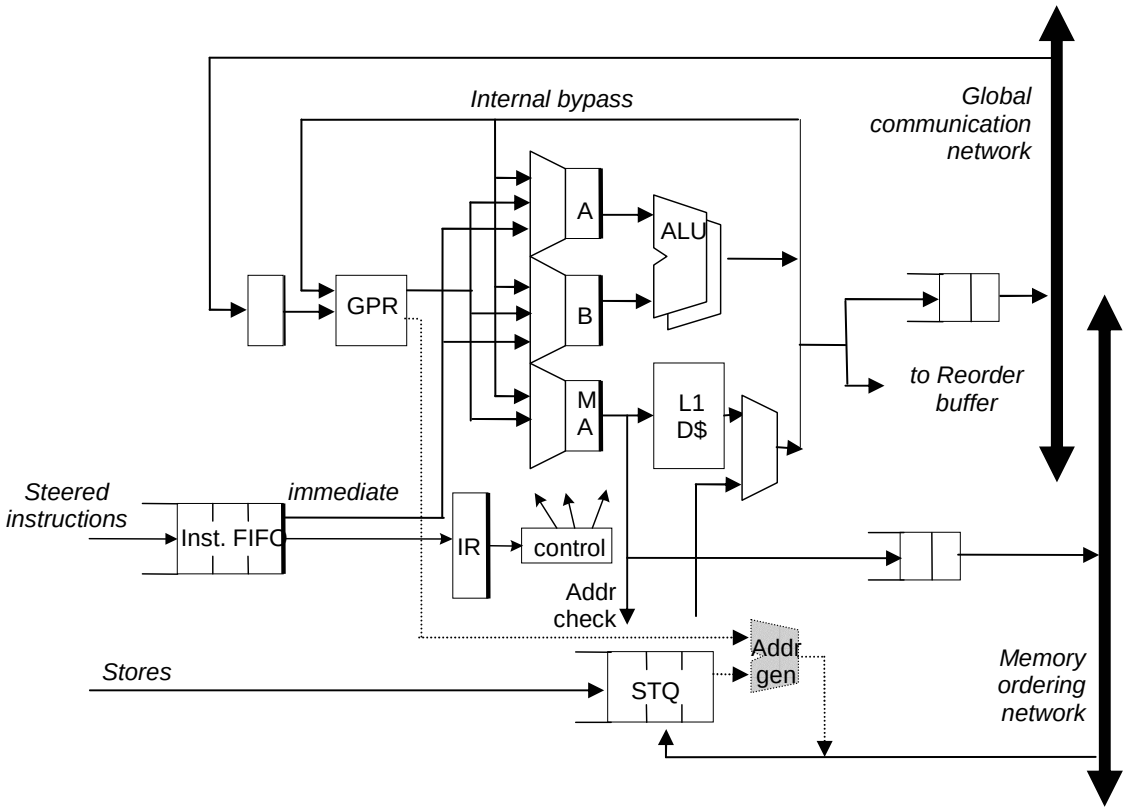



**Figure 3-5 Processing element internals**

The instruction at the head of the FIFO reads its GPR value (if available), and moves into the issue register (IR). If the GPR value is not available, the instruction waits in the IR until the value becomes available. Hence, the IR is like a single reservation station with a single data value entry. The three registers A, B, and MA implement a physical accumulator. A single physical accumulator is used for both integer and floating-point operand types. The computation results are written to the accumulator and also put into an internal buffer that arbitrates for the bypass network resources. If the number of entries in the buffer reaches a predefined high water mark, back pressure is sent to the dispatch stage to block further dispatches to the FIFO.

Note that the data cache is directly fed by the accumulator (the MA register in Figure 3-5) Therefore, the effective load-to-use latency of those V-ISA load instructions with zero offset is

reduced by a cycle. For those V-ISA loads that do need to calculate the address, there are two possibilities: (1) a separate address calculation instruction is created by the dynamic binary translator, (2) one of the ILDP instruction's `mode` bits informs the issue hardware to generate an address calculation micro-instruction dynamically. Due to the nature of the operand capture timing, load hit/miss speculation is not used at all.

In general, the functional units within a PE do not require pipelining. This will lead to simpler control logic within the PE and lower latencies for multi-cycle functional units. On the other hand, overlap between two adjacent strands is allowed in the PE. For example, if the last instruction in a strand is a load then the next strand can start issue without waiting for the load to finish. Otherwise, if the load misses in the data cache, it can block the issue of independent instructions in the next strand for an indefinite number of cycles. In any case, register dependences are correctly enforced through the coherent GPRs.

## 3.2.3 Data Cache

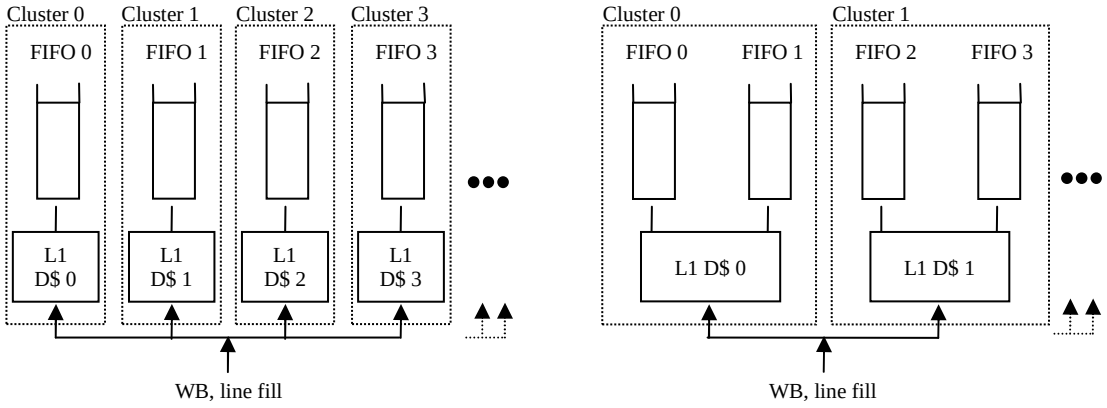
### 3.2.3.1 L1 D-cache Organization Options

Increasing the number of ports to an L1 D-cache is often very difficult [238]. Typically, conventional superscalar designs use either a fast L1 D-cache with a read port and a write port [108] or a slower L1 D-cache with two read ports and a write port [228]. Replicating the data cache [72] as in Figure 3-5 provides a third option: a combination of fast access time and multiple ports. However, fully replicating the data cache is a more expensive solution in terms of the transistor budget.

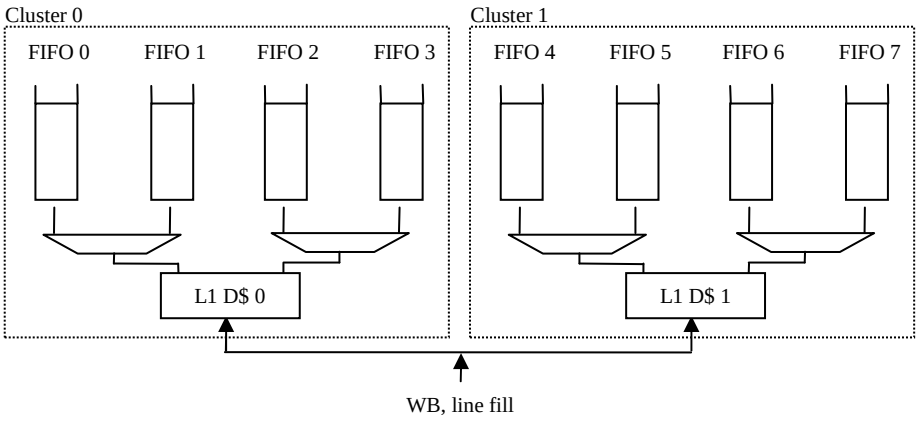
Other researches suggested dynamic partitioning as a way to provide a fast, multi-ported L1 D-cache without fully replicating the cache contents. In these proposals, replication is either completely prohibited [189] or allowed to some extent [190]. A key requirement is that load

instructions are steered to a data cache partition based on the calculated data address. Applying these ideas to the already distributed ILDP pipeline will result in another steering stage and hence, the speed advantage of the partitioned cache will probably be lost. In other words, if partitioning of the L1 D-cache is important, the overall microarchitecture should be built around the *memory* dependences, not as in the ILDP microarchitecture that was built around the *register* dependences.

Instead, more traditional cache sharing schemes can be used if the transistor budget (and static power consumption) is a concern. Figure 3-6 shows some of the possible data cache configurations. The last configuration, Figure 3-6(c), is used in the evaluation.




(a) Fully-replicated, single-ported D$ (b) Two FIFOs sharing a dual-ported D$




(c) Four FIFOs sharing a dual-ported D$.

**Figure 3-6 L1 D-cache organizations**

#### 3.2.3.2 Dynamic Memory Disambiguation

The distributed nature of the ILDP pipeline requires a distributed disambiguation solution. Unlike register dependences which are strictly enforced by the coherent copies of the physical register file, memory dependences are speculated in the ILDP microarchitecture. It will be shown in section 7.2 that the performance benefit of memory dependence speculation is substantial. Memory instructions are issued speculatively without waiting for all older memory instruction addresses are resolved, and then check the store and load queues later. If an ordering violation is found, the memory queues generate an ordering replay. A simple PC-based dependence predictor [165] is used to limit the number of ordering replays. The latency-critical store queue is replicated for each L1 D-cache read port while the less performance-critical load queue, used for detecting both load-to-load and store-to-load ordering violations, is shared by the multiple PEs.

Each replicated STQ allocates a store queue entry for each store request sent from the steering stage. One solution is to communicate the calculated store address to the replicated store queues over the memory ordering network. A possibly higher performance alternative would let each copy of the STQ perform store address calculation independently using an extra register read port and an address calculator (shown in grey in Figure 3-5). However this option mandates GPR assignment for the address register and hence, complicates the instruction format design and the ILDP instruction generator in the dynamic binary translator. Therefore, the first option – simple replicated STQs connected with a separate memory ordering network – is used in the thesis. A small number of shared busses are used in the evaluation.

## 3.3 Related Work

### 3.3.1 Complexity-Effective Superscalar Processor Designs

The ZS-1 [213] was an early superscalar design with instructions issuing simultaneously from two FIFOs, motivated by issue logic simplicity. The RS/6000 [13] used a similar design. The Alpha 21264 [131] replicated the register file in two execution clusters to reduce the number of read ports per register file. It takes an additional cycle for a value created in one cluster to be propagated to the other cluster. An instruction is issued to the cluster where its register value is available earlier than the other, from a shared issue window [80]. Note that here instructions are steered just prior to the *execution stage*, in contrast to the *dispatch stage* steering in the ILDP microarchitecture. This was the first attempt by a commercial microprocessor design to explicitly cope with on-chip global wire latencies and inspired many academic researchers. Current generation processors [129][152] [198][228] usually have smaller, multiple issue windows to meet aggressive clock cycle time goals. Many subsystems are heavily pipelined. As such, increased branch misprediction penalties and design complexity pose serious challenges.

On the cache side, the small number of ports to the L1 D-cache has been one of the primary factors that limit the issue width of superscalar processors. The Alpha 21164 used a fully replicated data cache to provide two read ports to the issue logic [72].

### 3.3.2 Complexity-Effective Research Proposals

Palacharla, Jouppi, and Smith [174] first defined complexity as the *critical path delay of a piece of logic* and identified register renaming, wakeup and select logic, and operand bypass network among others as key complexity-critical subsystems in an out-of-order superscalar design.

Instruction fetch bandwidth is a critical element of a high performance processor design. The fetch mechanisms introduced in [50][187][191] allows multiple sequential basic blocks to be fetched in the same cycle. Trace cache [177][179][196] is a well-known technique to dynamically reorganize basic blocks in their most frequent order. Trace caches can also be used to offload complex instruction decoding from the pipeline's critical path as in Intel Pentium 4 [109] and the canceled Fujitsu SPARC64 V design [65]. Other research proposals go even further by rescheduling instructions within dynamically collected traces. Fill-unit [85], DIF (dynamical instruction formatting) architecture [172], rePLay [77] belong to that category. In the thesis, dynamic code re-layout and external ISA instruction decoding (i.e., binary translation) are automatically performed by the superblock-based dynamic binary translator software. Therefore, (hardware) trace caches that are hardware-intensive and sensitive to transient traces [195] are not necessary. Instead, a fetch mechanism geared towards multiple sequential basic blocks is preferred.

Farkas, Chow, Jouppi, and Vranesic [78] studied the effect of the number of ports and size of the physical register file in out-of-order superscalar processors. To reduce the register file complexity, techniques based on interleaving [16][79][176][192][231][235] and hierarchy [16][38][55][183][192][243] were proposed. Many studies have been performed to find good instruction steering heuristics for clustered microarchitectures [18][21][42][43][82][175]. Complexity-effective instruction issue logic is another area where lots of research has been conducted [1][31][32][44][73][74][91][106][115][136][137][144][160][188][223]. Compared to these elaborate schemes on individual subsystems, the ILDP co-designed virtual machine paradigm provides a systematic way of reducing complexities in most key pipeline structures at the same time.

On the other hand, there is relatively little research on distributed data caches. Rakvic and Shen [190] proposed a distributed cache scheme that allows some "cachelets" to exist in multiple partitions. Here, it is not very hard to see that a steering scheme based on the calculated memory

addresses works better than another scheme based on the memory instruction's PC. In contrast, in Racunas and Patt's partitioned cache [189], a data cache line can only reside in one partition at a time. In the pipeline front-end, the memory instruction's PC is used to probe a steering predictor. If a memory instruction misses in the steered partition, the corresponding cache line is brought to the partition and an invalidation request for the cache line is broadcast to other partitions. In these proposals the entire microarchitecture is designed around the distributed L1 D-cache. As such, these distributed schemes do not fit well the ILDP pipeline that is distributed around register dependences.

# Chapter 4 Dynamic Binary Translation for ILDP

In the ILDP co-designed virtual machine system studied in the thesis, a simple and fast dynamic binary translation (DBT) system within the virtual machine monitor is used to provide binary compatibility with the existing software. As with the ILDP microarchitecture that is designed specifically to work with the accumulator-oriented ILDP I-ISA, the dynamic binary translation system presented here exploits the co-design nature of the paradigm to strike a good balance between software and hardware, performance and complexity.

In this chapter, the overall dynamic binary translation algorithm and related issues are discussed. First, dynamically constructed superblocks, the chosen units of translation, are explained along with the formation rules used in this research. Next, two of the key issues for a dynamic binary translation system, precise state maintenance and dynamic code expansion, are considered and the DBT policies to handle these issues are described. The actual translation algorithm for the ILDP I-ISA is presented next. Related code cache systems such as dynamic optimizers, binary translators, and co-designed virtual machine systems are listed at the end. The support mechanisms for efficient control transfers, one of the most important aspects of any code cache system, are described and evaluated separately in the next chapter.

## 4.1 Dynamic Binary Translation Framework

### 4.1.1 Operating Modes

The DBT mechanism in the thesis follows the prevalent “interpret/profile then translate/optimize” model used by most dynamic optimizers [14][26][33][34][45][58][62][139][159] and binary translators [9][20][46][60][68][143][193][202][232].

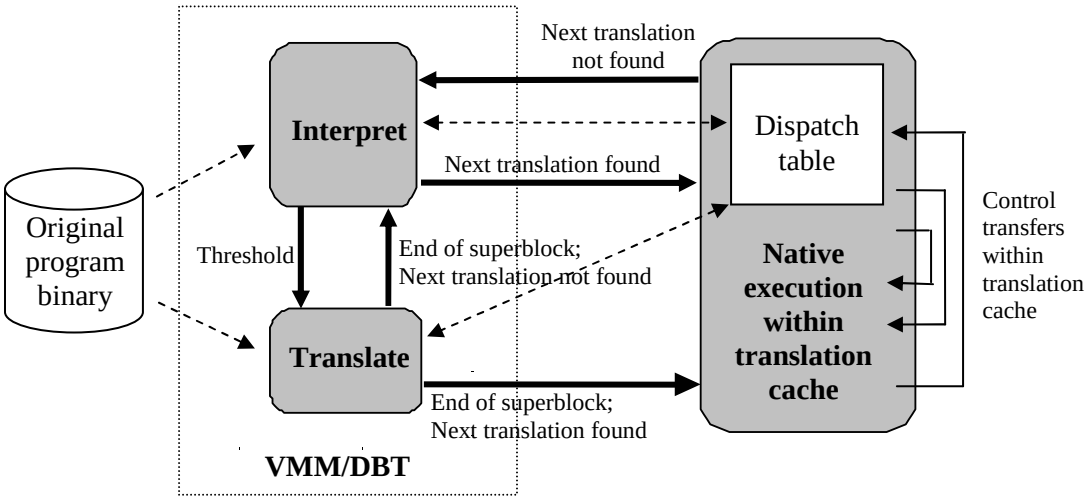


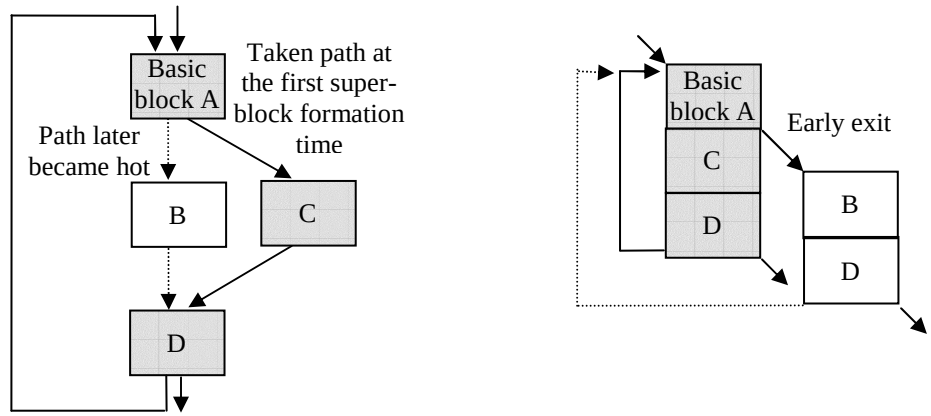

**Figure 4-1 Operating modes of the ILDP virtual machine system**

Figure 4-1 shows three main operating modes (interpretation, translation, and native execution within the translation cache) of the ILDP VM system. The VM system starts interpreting the given V-ISA (Alpha EV6 in the thesis) program. When a candidate start instruction for superblock, a straight-line code sequence with a single entry point and multiple exit points, is observed to execute a threshold number of times, the interpreted path is followed to generate a superblock. The superblock is translated into the ILDP I-ISA and placed in the translation cache (a code cache that holds ISA-translated program codes) in a concealed memory area. If later program control flow reaches an existing translation, the translated instructions are executed directly from the translation cache.

### 4.1.2 Translation Unit: Superblocks

As just noted, the basic unit of translation is a dynamically formed *superblock* [117]. Other than “early exit” points (shown in Figure 4-2(b)), there is a single path of control flow within a superblock. Practically all dynamic optimizers and binary translators use a superblock as their basic

optimization/translation unit because a superblock provides an easy-to-collect and highly repetitive series of basic blocks.[18] The nature of single flow of control also allows simpler implementation of optimization techniques. Figure 4-2 shows two dynamically collected superblocks chained together (Chaining is further explained in Chapter 5).




(a) Original program control flow (b) Chained superblocks

**Figure 4-2 A superblock formation example**

An important byproduct of this "hot" superblock collection method is that dynamic code re-layout is automatically performed; i.e., basic blocks that commonly occur in sequence dynamically are placed together statically (in the code cache), enhancing effective fetch bandwidth.

Like most other software-based translation/optimization frameworks, the code cache system used in the ILDP DBT allows only one superblock for a given V-ISA starting address. Hardware-based code cache systems such as trace caches sometimes allow multiple superblocks with the same

[18] A notable exception is IBM's DAISY which uses a tree-style VLIW group [69]. Nonetheless the tree-group in the DAISY system does not allow control-flow intersection within the translation unit.

starting address but different control flows to exist at the same time. An efficient chaining mechanism that allows fast transfer of control from a superblock to another is used for software-based frameworks to make up for the lack of partially redundant superblocks.

### 4.1.3 Superblock Formation Rules

The superblock formation algorithm used in the ILDP DBT system is a slightly modified version of the Most Recently Executed Tail (MRET) heuristic used in the HP Dynamo system [14][67]. Differences between the heuristics are marked with an asterisk (*) below. The DBT system starts interpreting the given V-ISA program. Over the course of interpretation, counters are maintained for the following possible superblock start candidate instructions:

- Targets of backward conditional branches
- Exit targets of existing translations
- Targets of register indirect jumps* (`JMP`/`JSR`/`RET` in the Alpha ISA)

If the number of times a static candidate instruction is executed reaches a predefined threshold, the interpreted path is followed to generate a superblock. This heuristic, a form of simple software speculation, is based on the observation that *when an instruction becomes hot, it is statistically likely that the very next sequence of executed instructions is also hot*. Superblock ending conditions are:

- A backward taken conditional branch is encountered
- Already collected instruction is found again (a cycle)
- A predefined maximum number of instructions is reached
- A register indirect jump* or trap instruction is encountered

A newly translated superblock, referred to simply as a *translation* hereafter, is placed in the translation cache. If later program control flow reaches an existing translation, the ILDP instructions in the translation are executed directly on the hardware.

## 4.2 Considerations for Dynamic Binary Translation

### 4.2.1 Maintaining Precise Architected State

There are two major issues in precise trap recovery in any dynamic binary translation (or optimization) systems that execute translated (or optimized) code other than the original program binary. Although theoretically the first problem of tracking V-ISA PC is part of the overall state maintenance problem, typically it is handled separately in actual implementations. As will be explained shortly, the ILDP DBT system does not change the order between translated instructions. This leads to a simple, fine-grain precise trap recovery model.

#### 4.2.1.1 Identifying the Trapping Instruction's Address

When a trap condition is encountered, control is transferred to the VMM. The address of the V-ISA instruction that generates trap must be identified. This is a non-trivial problem because in a code cache system the architected program counter is not used for actually executing the source binary code; rather an implementation program counter sequences through translated (or optimized) code in a code cache. Nonetheless, the trapping instruction's address, PC, is part of the V-ISA state, and hence must be recovered correctly. Some of the methods for locating the trapping instruction's address are:

- **Create checkpoints at translation boundaries.** When a trap is encountered, roll back the machine state to the last check point and interpret until the trap is reproduced as in Transmeta Crusoe [60].

- **Track the start address of the side table associated with a translation**. Use this address to locate a side table of potentially excepting instructions (PEIs) associated with the translation. A hardware counter is incremented for each PEI and is used as the index to the side table in case of a trap. ROAR dynamic optimization framework uses this method [173].

The second method is used in this research. A special load long immediate instruction that saves the starting address of the associated side table (as a 32-bit immediate) into the specified register is generated as the first instruction in any superblock.

#### 4.2.1.2 Restoring Architected State

All V-ISA state must be restored to the point of a trap. Again, there are largely two options – the state can either be maintained at translation unit boundaries (course-grain) or at each individual instruction level (fine-grain). As with identifying the trap PC, the ILDP DBT system maintains architected state at the fine-grain level. In contrast, other co-designed VM systems that maintain the V-ISA state only at translation unit boundaries, e.g., Transmeta Crusoe [128] and IBM BOA [9], need to buffer all state changes, including stores, before updating the architected state with a special commit instruction. With this method, finite pipeline buffer sizes act as the upper limit on translation unit size. More importantly, the coarse grain model can expose substantial interpretation overheads in pathological cases such as self-modifying codes [60].

The DBT system in the thesis does not reschedule instructions; hence values are produced in the same order as the original program. In the finalized ILDP I-ISA format, every instruction producing a V-ISA result specifies a destination GPR to maintain architected state. As a consequence, *all architected state updates are performed in the same order as the source program binary*. On the other hand, the translated instructions are executed out-of-order at the strand-level in the underlying ILDP microarchitecture. Nonetheless, their order (as in the translated code) is maintained by the reorder buffer and other hardware-based ordering maintenance mechanisms. A

separate register file, off the critical path, is maintained solely to keep the V-ISA GPR state for precise traps. Only the values needed for later computation, i.e., communication, are actually written to the performance-critical "working" physical registers in PEs. This is possible because the I-ISA format distinguishes destination register types (refer to section 2.3.3 for the description of the `mode` bits). Note that this scheme works in a different way from Transmeta Crusoe's working/shadow registers [99][140] where *all* register writes go to the working register file.

### 4.2.2 Suppressing Dynamic Code Size and Instruction Count Expansion

In general, in a system that supports a complete binary compatibility, dynamically removing source instructions that may affect the V-ISA state is very difficult or downright impossible. Combined with the fact that the translated I-ISA instruction contains more microarchitecture specific information, it is not too hard to see a tendency for effective code footprint expansion in a co-designed VM system. This is one of the most important aspects of any ISA translation systems because with more instructions to execute, it becomes harder to even match the performance of the native implementation.

In conjunction with the helper features provided by the ILDP I-ISA in section 2.3.2.2, the DBT system tries to limit code expansion via the following principles:

- **Use 16-bit format if possible**: Even though certain number of extra copy instructions are unavoidable due to the limited operand type combinations, their effects can be reduced through the use of a shorter instruction format. Also the use of 16-bit memory instructions (`LDQ16/LDL16/STQ16/STL16`) wherever possible contributes to suppress the code size expansion further.
- **Keep one-to-one instruction mappings as much as possible**: Most ILDP instruction formats use extra `mode` bits and 6-bit GPR identifiers so certain compromises in other bit-

fields are unavoidable. Typically the immediate values used in programs are smaller than the maximum values allowed by the bit field width defined by the V-ISA. In relatively rare cases where the immediate value exceeds the reduced size limit, either the source instruction is decomposed into multiple instructions (if possible) or a trap-to-interpreter instruction is used as a final safety net. When a memory instruction must be decomposed into an effective-address-calculation (EAC) instruction and an EAC-less memory instruction, a 16-bit memory instruction can be used to limit code size expansion for most of the time.

- **Exploit known V-ISA idioms**: To limit the number of extra copy instruction due to the lack of `GPR op Immediate` mode, the dynamic binary translator tries to exploit certain Alpha ISA idioms involving the constant zero register.
- **Use specialized instructions to enhance control transfer performance**: Translation of control transfer instructions has a fundamental impact on the overall performance; the conventional software-only translation method that converts a single control transfer instruction into an equivalent sequence of multiple instructions not only leads to code expansion but also negatively affects the related branch prediction performances, especially for indirect jumps. Use of specialized instruction and hardware support mechanisms are explained in section 5.2.

## 4.3 Binary Translation Algorithm

The major function of the translation process is identifying strands and re-mapping intra-strand communication values to accumulators. Because of the nature of dynamically constructed superblocks, there is no need for graph-traversing dependence analysis usually found in static compilers [169]. Below, major parts of the DBT algorithm are presented in order. Although the

algorithm is implemented in the research infrastructure as multiple passes of sequential scans, it is possible to combine certain passes to a single pass.

### 4.3.1 Superblock Construction

Once a superblock starting condition is met, candidate V-ISA instructions are interpreted and collected in a decoded form into a trace structure until the superblock ending condition is encountered. This intermediate representation (IR) format facilitates efficient dependence setup in the next step. Special care was taken to implement the decoder efficiently because it is one of the most time consuming processes as will be shown in section 7.3.2. On a related note, Bala, Duesterwald, and Banerjia [14] report the decoder takes the biggest chunk of their optimization framework binary.

When a superblock is being formed, a side table of V-ISA instruction addresses is generated for the superblock. This side table contains not only the source addresses of the potentially-excepting instructions but also the source target addresses, i.e., “early exit targets”, of conditional branches whose alternative control path targets were not found in the code cache at the time of translation. The side table performs double duty: first, it provides a storage area for the hardware/software co-designed trap PC identification mechanism. Second, when a conditional jump instruction (a place-holder for a conditional branch whose early exit target was not found at the time of the translation) transfers control to a patch code at run time, its source target address is found by indexing the side table with the conditional jump’s potentially excepting instruction (PEI) counter number (incremented by the hardware and stored in a special VMM register). If a translated target exists by then, the conditional jump is replaced with a conditional branch with a fixed offset value – a “patch” is performed. As was described before, the side table address is tracked by a special load long immediate instruction, the first instruction in every superblock.

These side tables are purely translation overhead and are seldom used. In the initial version of the DBT framework, side tables were placed next to their associated translations. After the arrangement was found to have non-trivial negative impact on the translated code's I-cache performance, a separate memory area was allocated for the side tables. Refer to Figure 6-5 for the hidden memory map of the DBT system in the thesis.

### 4.3.2 Inter-Instruction Dependence Setup

Because of the nature of superblock-based translation algorithm, it is not practical to separate static global values from communication global values. Instead, a single pass linear scan of the superblock yields register dependence and usage information. Here, live-in values on superblock entry and live-out values on *all* early exits are considered global as well as communication global values.

One possible alternative is to *not* consider local values as live-out global values at the early exit points. This will lead to somewhat higher probability of accumulator assignments within the translation. However, early exits do happen with non-trivial frequencies (as shown in section 5.3.2) and there must be a safety net mechanism for maintaining the accumulator values and their mappings for that. Use of a "fix-up" code when an early exit is taken is one way to achieve that goal. With this mechanism in place, some live-in values can be assigned accumulators in a newly chained superblock (basic blocks B and D in Figure 4-2). However, this on-demand fixing scheme not only introduces added complexity of fix-up code generation algorithm but also increases the amount of work the dispatch table lookup code has to do before transferring control appropriately.

For this reason, a rather conservative live-out policy where all potential early exits are considered as superblock boundaries is used. This policy of maintaining live-in and live-out values

in GPRs results in lower utilization of accumulators. Following are the important register value categories used in the DBT.

- **No-user**: Outputs register value not used before being overwritten. An instruction whose output value is not used naturally ends a strand.
- **Local**: Outputs register value used only once before being overwritten in the same superblock. These are candidates for assignment to accumulators.
- **Temp**: Values passed between two decomposed instructions, e.g., conditional moves and some memory instructions. These are assigned to accumulators.
- **Live-in global**: Input register values that are live on superblock entry; assigned to GPRs.
- **Live-out global**: Output register values live on superblock exit; assigned to GPRs.
- **Communication global**: Register values used more than once before being overwritten in the same superblock; assigned to GPRs. The first use of a communication global value is considered as a local, i.e. the first user instruction continues the strand.
- **Spill global**: (a) If an instruction has two local input registers, one is made a spill global because the I-ISA does not allow two different accumulators in the same instruction. (b) If a strand has to be terminated to free an accumulator, a local value is converted to a spill global.

Figure 4-3 shows the output register value types in the translated superblocks. On average, more than 25% of dynamic instructions have a global output value (live-out and communication globals). These are the global values that should be written to the latency critical physical register file. The rest of the produced register values go to the local accumulator. All produced output values are also used to update the architected register file, off the critical path of the processor pipeline. The notation “local → global” represents values that are used only once in the superblock but need to be saved to a GPR before an early exit. The notation “no user → global” is similar. Once these

values are included, the total percentage of instructions that have global output values rises to about 40%.

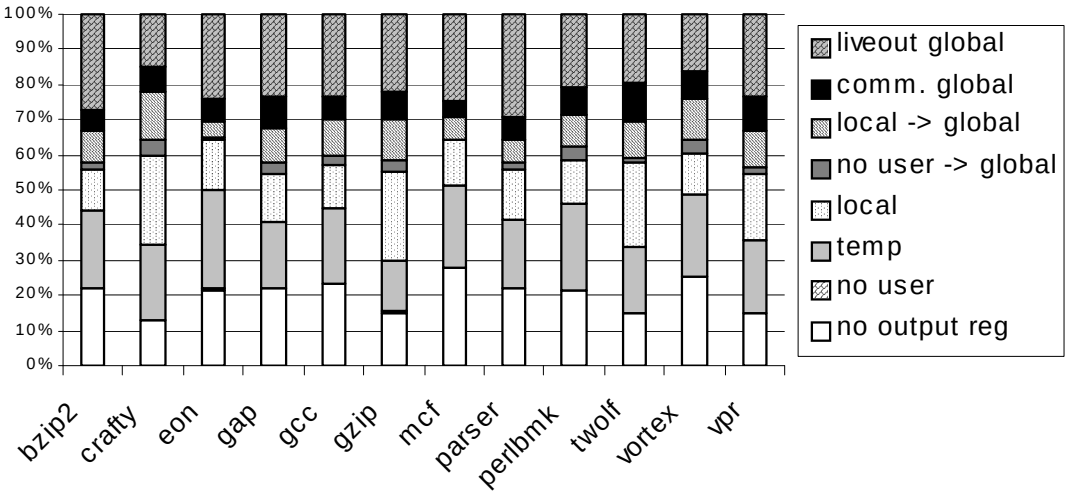



**Figure 4-3 Output register types in superblock-based dynamic binary translation**

Those statistics are in contrast to the earlier program trace in section 2.1.4 where only about 20% of instructions were found to produce global output values from program traces (e.g., register values do not need to be saved at superblock boundaries).

### 4.3.3 Strand Identification

Based on dependence and input register usage patterns, instructions are scanned and a strand number is assigned to each instruction. The translator uses an unlimited number of strands that are later assigned to a finite number of accumulators. Here temp usage is treated the same way as local. If an instruction has:

- Zero local input registers: a strand is started and a new strand number is assigned to it. Furthermore, instructions with two global input registers are broken into two accumulator instructions – a copy-GPR-to-accumulator instruction and a translated source instruction

that uses the (local) result of the copy as an input. The copy instruction initiates a strand, and the source instruction now has one local input register (and is handled accordingly).

- One local input register: assigned the same strand number as the instruction producing the local value.
- Two local input registers: a heuristic is needed to decide which strand number to be assigned. For store instructions, the address register is always selected as a temp value. For other instructions, if one of the input registers is a temp, then the temp producer's strand number is assigned to the instruction. Otherwise the number assigned corresponds to the longer strand up to that point (length is determined by instruction count).

Once all strands are identified within the superblock, the last instructions in each strand are marked as end-of-strand.

### 4.3.4 Accumulator Allocation

The strand numbers are converted to finite accumulator numbers. Instead of using a traditional graph coloring heuristic [169] to assign accumulator numbers to strands, a simple linear-scan heuristic is used. When the translator runs out of accumulators, a live strand is chosen for termination and the accumulator is freed. This is done by changing the new end-of-strand instruction's output register type from accumulator to GPR at the strand termination point, and changing the new start-of-strand instruction's input register type to GPR. A simple, greedy heuristic that selects the latest alive strand is used.

Although the ILDP I-ISA format allows for up to 8 logical accumulators, a co-designed VM system can choose to have smaller number of logical accumulators. In any case, the ILDP microarchitecture has to provide at least the same number of FIFOs as the number of logical accumulators used. This is to prevent a live-lock situation.

### 4.3.5 ILDP Instruction Generation

Once the accumulator assignment is finished, the source instructions in IR format are converted to corresponding ILDP instructions. In the process, certain helper instructions such as the load long immediate for side table tracking, an optional unconditional branch at the end (either to a shared dispatch code or an existing translation), 16-bit ILDP NOPs for aligning translations at page boundaries, are introduced. As with the source instruction decoder, the ILDP instruction generator extensively utilizes shift and mask operations to achieve high speed and a small code footprint.

## 4.4 Related Work

### 4.4.1 Dynamic Binary Translators and Optimizers

Most dynamic binary translators translate from one existing instruction set to another, with code portability as the primary goal. The software-based ISA translators such as DEC FX!32 [46], Sun WABI [111], HP Shogun [143] and Aries [244], Strata [202], UQDBT [232], Intel IA-32 EL [20] and dynamic Java compilers such as IBM Jalapeno [11] and Sun HotSpot JVMs [155] belong to this category. Typically code optimizations are implemented, many of them ISA-specific, and the performance goal is one of reducing losses; i.e., to come reasonably close to native ISA execution. Furthermore most existing binary translators are focused on application binary interface (ABI) translation rather than full ISA translation, as is the case with co-designed VM systems. Architecture simulators are closely related to binary translation systems – high-performance simulators such as Embra [240] and Shade [48] typically translate and cache the frequently executed portions of the simulated program in a code cache.

Dynamic optimization (without ISA translation) can be performed by either software as in HP Dynamo [15][33], DynamoRIO [34], Mojo [45], Wiggins/Redstone [58], DELI [62], Kistler and

Franz [139], Tamches and Miller [225] or a specialized hardware-based optimization framework such as rePLay [77], fill unit [85][88][122], ROAR [158][173]. The primary objective in this work is performance improvement, and therefore some of the techniques used, e.g. code re-layout and optimized code caching, are related to the DBT system in the thesis. Although performance profiling/feedback [57][104][157][200] and program phase detection [66] mechanisms are important parts of a dynamic optimization framework, they are beyond the scope of the thesis research.

### 4.4.2 Co-Designed Virtual Machines

Co-designed virtual machines were studied in the IBM DAISY [68][69][70] and BOA [9][95] projects and are implemented in the Transmeta Crusoe processor [60][99][140], all of which targeted VLIW implementations. In contrast, the research here is targeted at a simple form of dynamic superscalar implementation. Rather than trying to maximize instruction-level parallelism on a static VLIW microarchitecture using aggressive optimization techniques, the ILDP DBT system simply identifies inter-instruction dependences and encodes the dependence information as accumulator assignments without changing the original program order. Maintaining the original instruction order greatly simplifies precise trap recovery. All in all, the ILDP co-designed virtual machine system in the thesis has less binary translation overheads and has a good chance of balancing the strengths of hardware and software better.

# Chapter 5 Efficient Control Transfers within a Code Cache System

Most dynamic binary optimizers and translators, including the one in the thesis, first map the source binary code into superblocks then optimizer/translate and place them in a code cache for repeated execution on the target platform. When executing within a superblock, performance is enhanced, both because of optimizations that may have been done and because of straight-line instruction fetching that naturally occurs. For the ILDP dynamic binary translation system that does not employ aggressive optimization techniques, the automatic code re-layout is a major asset to offset the interpretation and translation overheads. However, when making transitions from one cached superblock to another, there is a potential for performance loss. For example, if a dispatch table lookup mechanism must be invoked before each new superblock can be entered, then all performance gains would likely be lost (and then some). One commonly used optimization is to "chain" superblocks together so that one can immediately branch to the next, but this method only works with direct branches. For indirect jumps, the problem is more difficult and remains a problem in many systems.

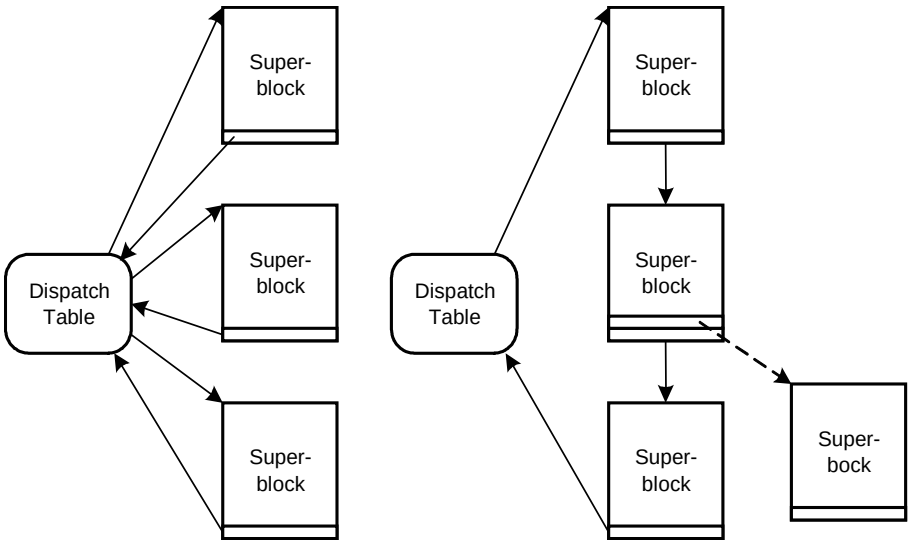
This chapter looks at the popular software-based chaining method used in many code cache systems and identifies where the performance problems are. Next, hardware/software co-designed support mechanisms are proposed for efficient control transfers among superblocks being held in a code cache. To separate the effect of code re-layout and specialized control transfer support mechanisms from the ISA translation, an identity-translation system is used to evaluate the support features within the chapter. This is also because that the subject of efficient chaining mechanism is a general problem for all types of code cache systems, be it a dynamic binary translator or a optimizer.

## 5.1 Superblock Chaining

When a code cache system forms a superblock, it typically places an entry in a dispatch table, i.e. a hash table that maps *source* binary *program counter* values (SPCs) to *translated* binary *program counter* values (TPCs). As a bare minimum, a code cache system can be made to transfer control between cached superblocks by consulting the dispatch table every time a branch or jump instruction is encountered. If there is a hit, control is transferred to the mapped superblock in the code cache via the TPC. Similarly, when the end of a superblock is reached, the dispatch table is accessed to find the next superblock (if it exists). Typically, this lookup would require several instructions, including at least two memory accesses, ending in an indirect jump. Obviously this will lead to substantial overhead. At that point if there is a miss in the dispatch table, control is passed back to the interpreter. Initially program execution switches between the interpreter and the code cache frequently, but eventually the program will be executed almost entirely within the code cache.

### 5.1.1 Chaining for Direct Branches

Fortunately, direct branches, either conditional or unconditional, are relatively easy to optimize because their (taken) target addresses do not change during program execution. Superblocks can be chained together so that a direct branch from one superblock to another can be made directly without relying on SPC-to-TPC mapping. Here chaining for a source branch instruction is simply a matter of generating a new displacement immediate field (with possible reversal of branch direction) for the corresponding branch instruction(s) in the mapped superblock. This type of chaining is commonly done in systems that use code caches and is illustrated in Figure 5-1.




(a) Without chaining (b) With chaining

**Figure 5-1 Control transfers among superblocks**

### 5.1.2 Conventional Chaining Method for Indirect Jumps

For indirect transfers, however, the problem is more difficult. Register-indirect jumps have their target addresses stored in a register, and the register value can change over the program's execution. Furthermore, this address is an SPC value, not a TPC value. This means that the original jump target address being held in a register must be translated every time the indirect jump instruction is *executed*, not only when it is *translated*. The most straightforward solution is to consult the dispatch table for every indirect jump. Hence indirect jumps still have a significant performance cost.

To save table lookup overhead for each and every indirect jump, many dynamic optimizers/ translators [14][33][68][240] implement a form of software-based jump target prediction. Typically a sequence of instructions compares the indirect target SPC held in a register against an embedded translation-time target SPC. A match indicates a correct "prediction" and the inlined direct branch

instruction is executed; if not, the code jumps to the shared (slow) dispatch code. Although the example in Figure 5-2 shows three sequential predictions, many systems use only one prediction.

```
If Rx == #addr_1 goto #target_1
Else if Rx == #addr_2 goto #target_2
Else if Rx == #addr_3 goto #target_3
Else hash_lookup(Rx); do it the slow way
```

**Figure 5-2 A code sequence that perform indirect jump target comparison**

The software prediction method is of limited value, however. First, if the target address is not one of the selected addresses, then time is wasted by testing the possibilities, and the dispatch table lookup has to be performed anyway. The performance cost of a misprediction is high. Second, there are a number of indirect jumps that are not very predictable using this method. For example procedure returns often have a number of call sites, and therefore a number of changing targets. Bruening, Duesterwald, and Amarasinghe [33] identify this indirect jump problem as *the* highest overhead in a code cache system and report that a hash table lookup takes 15 instructions, while the software comparison of the target of an indirect jump takes 6 instructions in the x86 ISA.

Previous code cache systems considered the software prediction technique (along with partial inlining of jump target code) as an *optimization*. However, over the course of the research I have found that this technique is rather a *performance limiter*, especially for returns, and started looking for alternative methods.

# 5.2 Supports for Efficient Code Cache Control Transfers

## 5.2.1 Software-based Jump Chaining Methods

In Figure 5-3, three different software-based indirect jump chaining options are depicted for an indirect function call instruction (e.g. JSR in the Alpha ISA). The dispatch table lookup code is shown in gray. These software-based methods affect the underlying hardware branch predictor behavior in a negative way as they convert a single indirect jump instruction to a sequence of codes including multiple control transfer instructions. In Figure 5-3(a), an indirect jump is converted to an unconditional branch to the shared dispatch code. The target address prediction rate of the register-indirect jump in the dispatch code is expected to be very poor because all indirect jumps lead to the same dispatch code and a single BTB entry is required to provide all the target addresses.

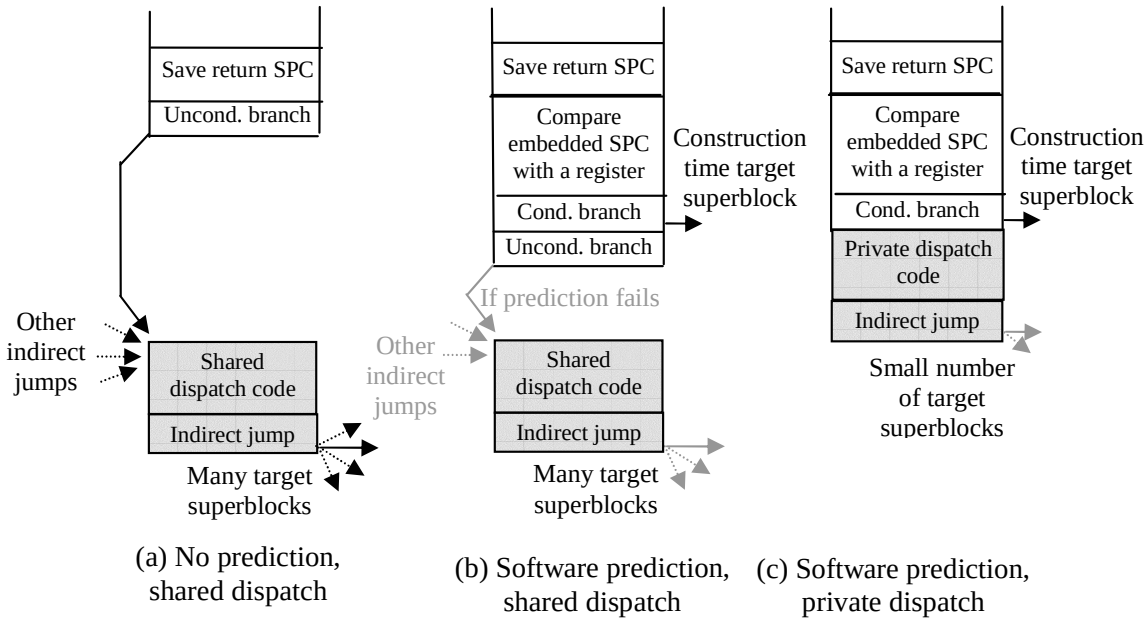



**Figure 5-3 Software-based jump chaining methods**

The conventional indirect jump chaining method based on software prediction is shown in Figure 5-3(b). Here, the compare-and-branch code reduces the number of times the shared dispatch code is executed. Hence, the pressure on the BTB entry for the indirect jump in the dispatch code is somewhat reduced. However, many times the software prediction is incorrect and in that case two mispredictions can happen – one by the conditional branch in the compare-and-branch code, another by the indirect jump in the dispatch code. The conditional branch has less impact because the branch predictor will eventually be trained to predict the branch as not-taken.

An alternative software method (Figure 5-3(c)) is proposed by the thesis: replicate the dispatch code after every register-indirect jump, thereby allowing "private" target address prediction in case the superblock construction-time prediction fails. This way the number of mispredictions by the indirect jump in the dispatch code is reduced. This option trades off superblock size, which leads to increased I-cache pressure, for a better target address prediction rate. The private dispatch code concept is similar to the one used in threaded code *interpreters* [76]. However, I am unaware of any previous proposal or existing system that applies this "threaded" technique to a dynamically translated/optimized *code cache* system.

### 5.2.2 Jump Target-address Lookup Table

One way to avoid the expensive dispatch table lookup almost entirely is to maintain a hardware cache of dispatch table entries. This specialized chaining support feature is called the Jump Target-address Lookup Table (JTLT) in the thesis. The JTLT is maintained by the code cache manager and always provides a correct translated address if there is a hit. The concept is similar to the software-managed TLBs used in virtual memory systems [121].

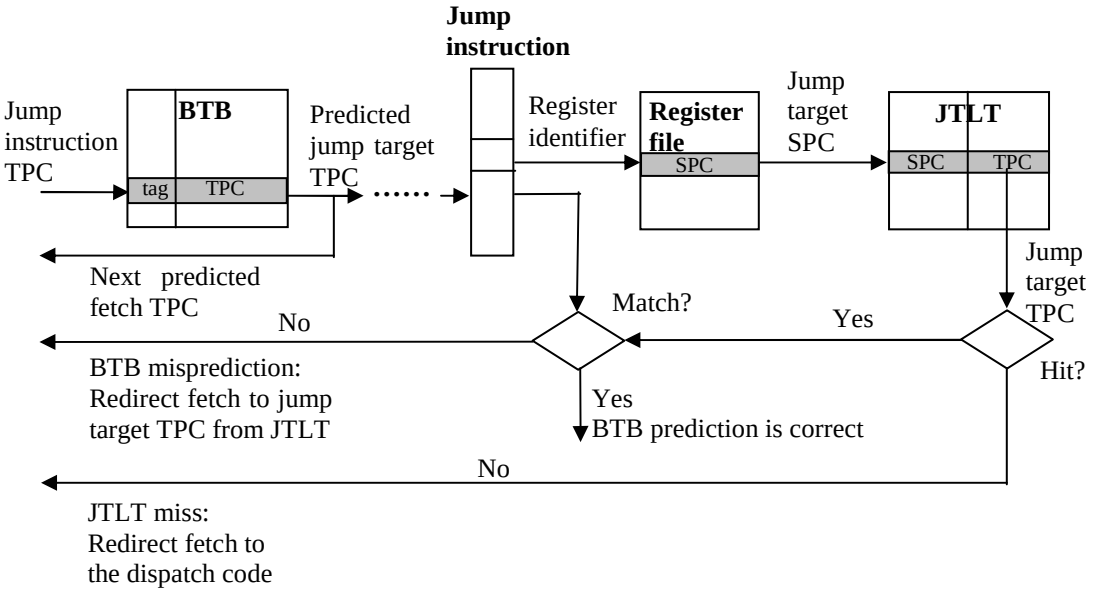



**Figure 5-4 Jump target-address lookup table**

Figure 5-4 shows how a JTLT can be used in conjunction with a BTB as a checker/predictor pair. An indirect jump instruction's target TPC is predicted with the normal BTB. This predicted target address flows through the pipeline with the jump instruction itself, just as in any other prediction mechanisms. When the jump instruction reads the target SPC from its register, the JTLT is searched. If there is a hit at an entry that matches the predicted TPC, the prediction is correct. There are two ways of mispredicting. First, the JTLT itself may miss. In that case, the hardware alone cannot provide the correct target TPC. The jump is not taken and the next sequential instruction, a branch to the dispatch code, is executed. Second, even if a JTLT entry is found, its TPC may be different from the one provided by the BTB. This is a BTB misprediction and fetch is redirected to the TPC from the JTLT.

If the JTLT is used, a register-indirect jump is not translated to a compare-and-branch code sequence and remains as an indirect jump. This suppresses dynamic instruction count expansion

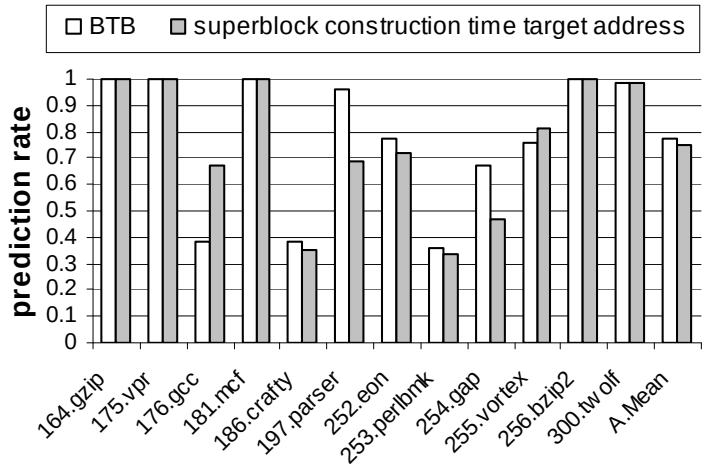
found in software-based prediction techniques. However, the BTB/JTLT pair also has a couple of weaknesses. First, it requires on-chip storage space. In the Transmeta Crusoe processor a 256-entry "TLB" [99] is believed to be used for this purpose. Assuming a 32-bit SPC and a 16-bit TPC, a 256-entry fully-associative JTLT uses 1.5KB of associative memory storage. Second, the BTB/JTLT pair does not provide a highly accurate return address stack (RAS) [126] type prediction capability for return instructions.

### 5.2.3 Dual-address Return Address Stack

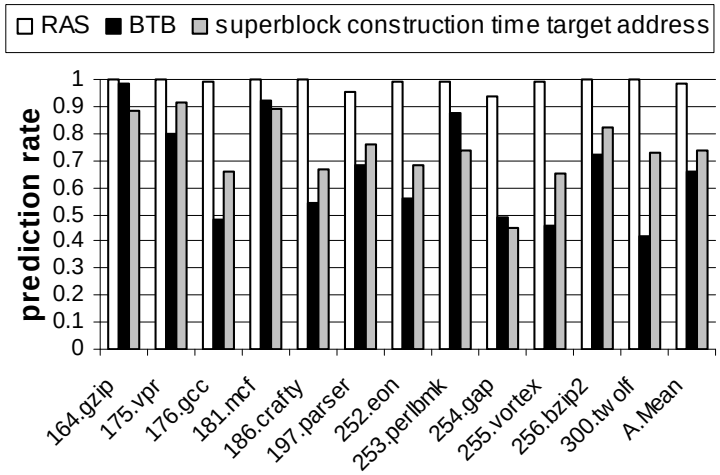
Most modern processors use a RAS mechanism, which can predict a return instruction's target address very accurately. In a dynamically managed code cache system, however, a conventional RAS cannot be utilized. First, typically the next instruction in the code cache after a function call instruction (or an equivalent code sequence) is not the return target instruction. Therefore there is no simple way for a RAS to find and push the return address (the function call instruction's `PC + 4` in the Alpha ISA). Second, even if the *TPC* of the return target instruction were pushed (and popped for an address prediction) by some means, there is no easy way to verify if the RAS prediction was correct. This is because the saved return target address is an *SPC*.

This inability to use a conventional hardware RAS leads to substantial performance loss, as can be seen in Figure 5-5. It can be seen from Figure 5-5(a) that for non-return indirect jumps, the conventional software-based prediction technique (described in section 5.1.2) is almost as good as the dynamically trained BTB. However, returns are a totally different story. Figure 5-5(b) shows that, compared to a RAS, a BTB (used instead of a RAS) and the software prediction technique result in 34% and 25% more mispredictions, respectively. Interestingly, the BTB performance is actually *lower* than the quasi-static software prediction (due to trashing) in Figure 5-5(b).

To solve this performance problem, some even proposed saving the return TPC into the register [202]; however doing so breaks the precise state maintenance model.



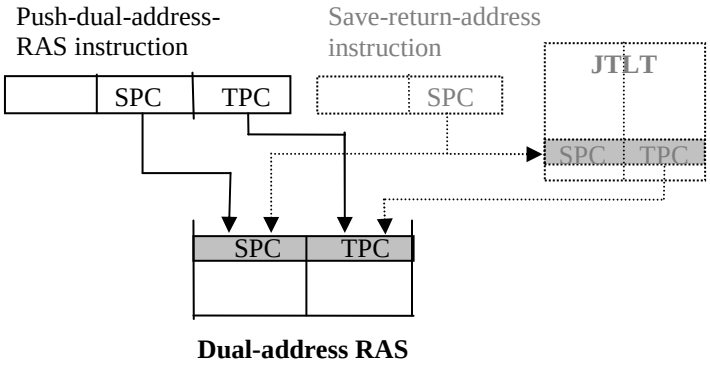

(a) Non-return indirect jump



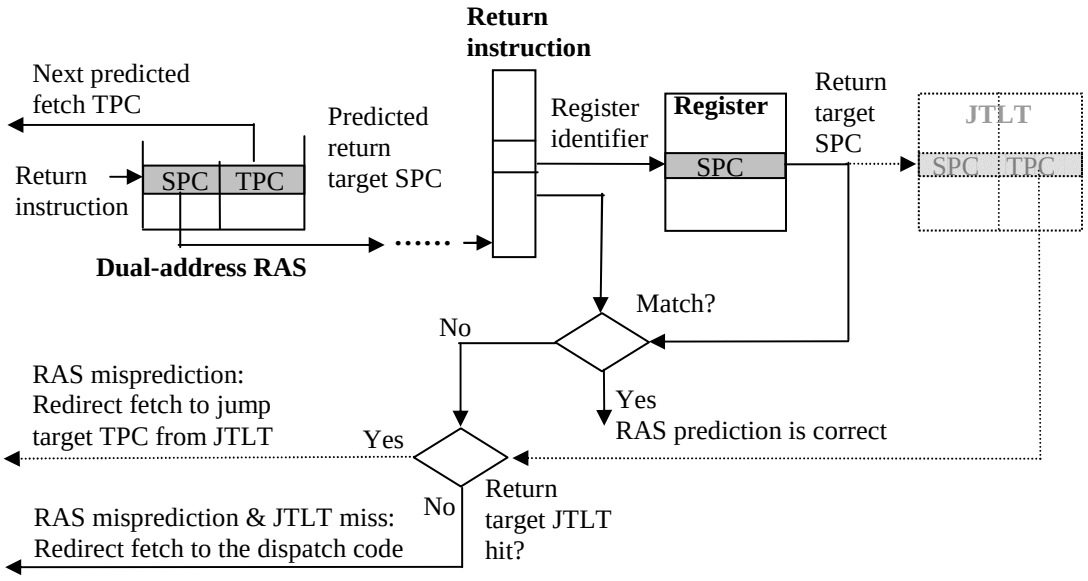

(b) Return

**Figure 5-5 Indirect jump target address prediction rates**

A specialized RAS mechanism that contains an address *pair*, consisting of a return address SPC and its corresponding TPC is shown in Figure 5-6.




(a) RAS push




(b) RAS pop and prediction check

**Figure 5-6 Dual-address return address stack**

There are two ways to push a pair of addresses onto the dual address RAS. The first option is to use a special *push-dual-address-RAS* instruction that pushes both return addresses. Finding the

return target SPC at superblock construction time is simple (`PC + 4` of the function call instruction). If the corresponding TPC is not found at superblock construction time, an invalid address is written in the TPC field. Later when the return target superblock is constructed, the invalid address is replaced with a valid TPC.

Another way to form a return address pair is to consult the JTLT when the original return target SPC is pushed. With the JTLT, the return target TPC does not need to be embedded as in the *push-dual-address-RAS* instruction. When an instruction that saves a return address is encountered, the JTLT is searched for a matching TPC. If a match is found, a pair of return addresses are formed and pushed onto the dual-address RAS. Note that a conventional load (short) immediate instruction pair that saves a return SPC can not be used in this case. Typically, a load immediate instruction does not contain a hint to push the RAS. Instead, a special *save-return-address* instruction that contains only the SPC can be used.

When a return instruction is fetched, the next fetch address is predicted with the popped TPC. The SPC part of the pair flows down the pipeline with the return instruction and is compared to the register value. If the two values do not match, a RAS misprediction is detected and fetch needs to be redirected. If a JTLT is also used, it can be relied upon to provide the correct target TPC. Otherwise, fetch redirection is accomplished by a branch to the dispatch code. Note that the semantics of this return instruction are slightly different from the conventional definition – a conventional return always jumps to the target. Here, if the return prediction is not correct, the next sequential instruction is executed (or the JTLT sets the next TPC if used and is hit).

With this specialized RAS, a return is not converted to the compare-and-branch sequence. The dual address RAS dramatically improves the prediction rate for returns and removes many extra instructions that would have been generated for a single return instruction, like the JTLT does.

### 5.2.4 Summary of Special Instructions and Jump Chaining Methods

Table 5-1 summarizes the special control transfer support instructions in the chapter. It should be noted that a code cache system can judiciously choose a subset, based on performance requirements, implementation constraints and hardware budget.

**Table 5-1 Special instructions to reduce register indirect jump chaining overhead**

| Category | Instruction | Description |
|---|---|---|
| Load long immediate | Save-return-address | Saves the immediate value (return target SPC) to a register. Also gives a hint to the prediction hardware to form a pair of return addresses using the JTLT and push it onto the dual-address RAS. |
| | Push-dual-address-RAS | Contains two immediate values. Saves the first immediate value (return target SPC) into a register. Gives a hint to the prediction hardware to push both the first and the second (return target TPC) immediate values onto the dual-address RAS. |
| Conditional jump | Predicted-indirect-jump | Conditionally jumps using a register (contains jump target SPC). If there is a JTLT miss, does not jump; the next sequential instruction, an unconditional branch, will branch to the dispatch code. |
| | Predicted-return | Conditionally jumps using a register (contains return target SPC). Gives a hint to the prediction hardware to pop the return target TPC from the dual-address RAS. If the RAS prediction is incorrect, either (a) does not jump or (b) jumps to the target TPC from the JTLT if the optional JTLT is used. |

Table 5-2 summarizes the register-indirect chaining methods that are evaluated in this chapter. Each is named via a pair of terms separated by a period; these are the prediction method for non-return jumps and for returns, respectively. For example, in the *sw_pred.ras* method, software prediction is used for non-return jumps and the dual-address RAS is used for returns.

**Table 5-2 Summary of jump chaining methods**

| Indirect jump chaining method | Description | | |
|---|---|---|---|
| | Execute dispatch code? | Return prediction mechanism | Related figures |
| *No_pred.no_pred* | Always | BTB (an jump in dispatch code) | Figure 5-3(a) |
| *Sw_pred.sw_pred* | When SW prediction failed | BTB (a conditional branch plus an jump in dispatch code) | Figure 5-3(b), (c) |
| *Sw_pred.ras* | When SW prediction failed (non-return jumps)<br>When RAS prediction failed (returns) | RAS | Figure 5-3(b), Figure 5-6 |
| *Jtlt.jtlt* | When JTLT missed | BTB | Figure 5-4 |
| *Jtlt.ras* | When JTLT missed | RAS | Figure 5-4, Figure 5-6 |

# 5.3 Comparisons of Superblock Chaining Methods

## 5.3.1 Identity Translation: Separating the ISA Effect from Chaining

In general, superblock chaining methods apply to broader range of code cache systems, not just to the co-design VM system studied in this thesis. Besides, it makes sense to separate the effect of ISA translation from the chaining mechanisms so we can understand both mechanisms better. To serve this goal, an "identity translation" (where the Alpha ISA is mapped onto itself and no other translation/optimization is performed) mechanism was built on top of an early version of the baseline simulator. This baseline configuration without code caching is referred to as *original*. Essentially the identity translator is a stripped-down version of the ILDP DBT mechanism (described in section 6.4.1). More details on the baseline simulator used in this chapter are in the appendix at the end.

On the other hand, it should be noted that a code caching system without any optimization techniques (other than the automatic code re-layout) is in itself an important design point when strict binary compatibility is required.

### 5.3.2 Superblock Characteristics

**Table 5-3 General superblock characteristics**

| Benchmark | No. of dynamic source instructions | % of instructions executed in code cache | Superblock completion rate | Average number of instructions between taken control transfer instructions | |
|---|---|---|---|---|---|
| | | | | original | code cache |
| *164.gzip* | 3.25 billion | 0.9999 | 0.76 | 13.6 | 27.3 |
| *175.vpr* | 1.44 billion | 0.9996 | 0.58 | 13.7 | 28.2 |
| *176.gcc* | 1.77 billion | 0.9938 | 0.73 | 9.7 | 19.1 |
| *181.mcf* | 210 million | 0.9989 | 0.70 | 8.7 | 10.1 |
| *186.crafty* | 4.07 billion | 0.9995 | 0.55 | 12.7 | 30.0 |
| *197.parser* | 3.92 billion | 0.9996 | 0.77 | 8.1 | 13.9 |
| *252.eon* | 89.7 million | 0.9899 | 0.88 | 14.3 | 23.6 |
| *253.perlbmk* | 3.69 billion | 0.9998 | 0.91 | 10.4 | 18.8 |
| *254.gap* | 1.11 billion | 0.9978 | 0.80 | 9.9 | 19.2 |
| *255.vortex* | 3.74 billion | 0.9991 | 0.91 | 10.7 | 36.3 |
| *256.bzip2* | 4.16 billion | 0.9999 | 0.96 | 14.0 | 20.1 |
| *300.twolf* | 238 million | 0.9946 | 0.61 | 14.5 | 23.2 |
| **Average** | | **0.9977** | **0.76** | **11.7** | **22.5** |

First consider characteristics of cached code in Table 5-3. From the 3rd column of the table, it is apparent that all benchmark programs almost always execute within the code cache – even for these very short (in real terms) benchmark runs. On the other hand, it can be seen from the superblock completion rates in the 4th column that early exists are not that infrequent. This suggests that a fast and efficient chaining support for conditional branches can be important. The ILDP I-ISA provides special place-holder instructions (described in section 2.3.3.3) for those early exits.

Of primary interest in this data is the average number of instructions between taken control transfer instructions in the last two columns. On average, dynamic superblock caching achieves about a two-fold increase compared to original program execution.

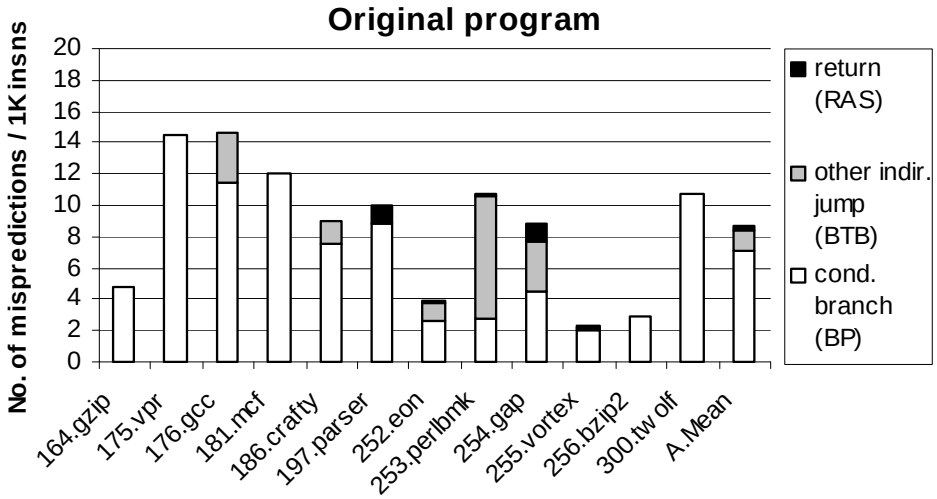
**Table 5-4 Dynamic instruction count expansion rate**

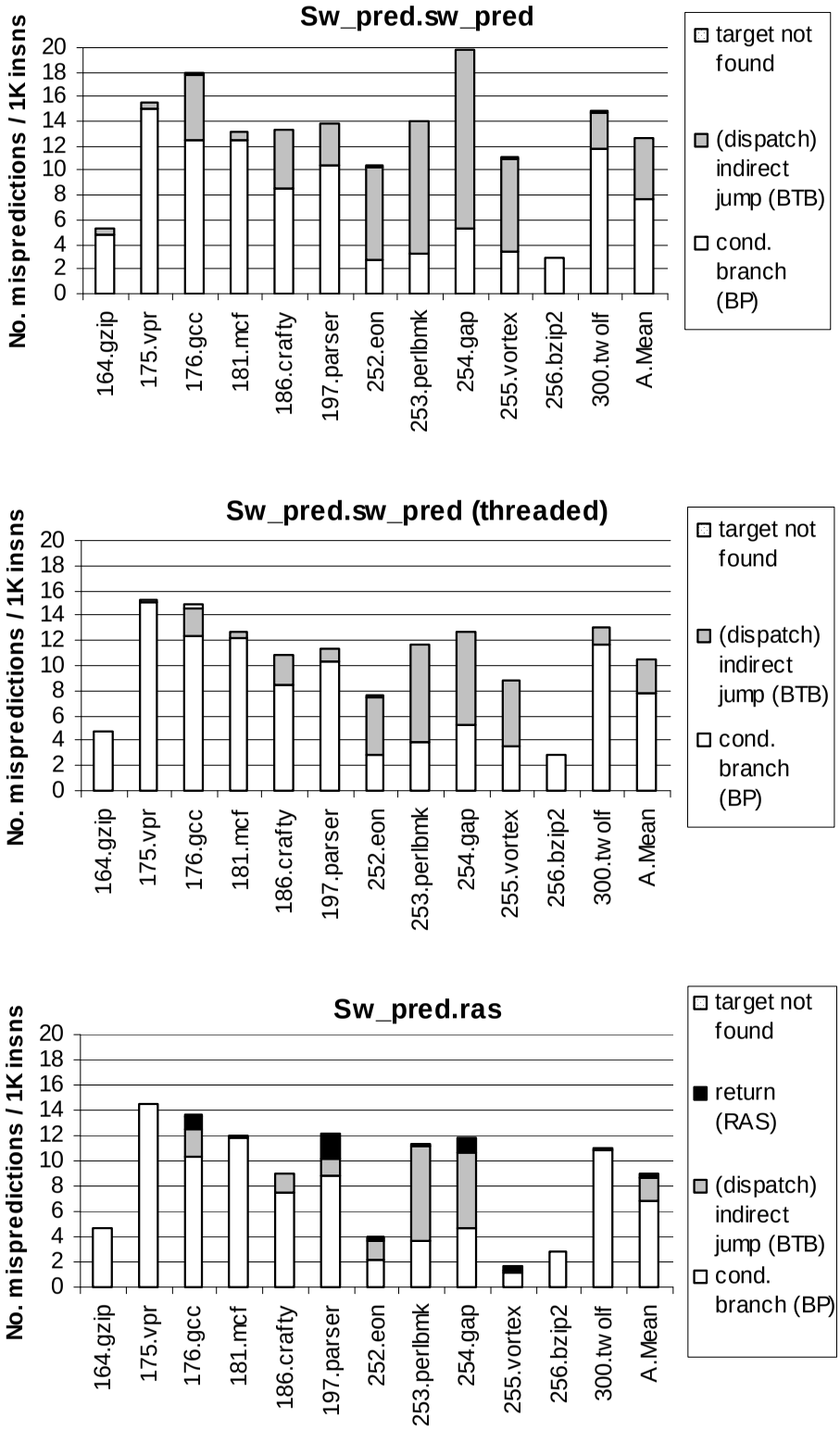
| Benchmark | *No_pred.no_pred* | *Sw_pred.sw_pred* | *Sw_pred.ras* | *Jtlt.jtlt* | *Jtlt.ras* |
|---|---|---|---|---|---|
| *164.gzip* | 1.149 | 1.035 | 1.000 | 1.000 | 1.000 |
| *175.vpr* | 1.182 | 1.037 | 1.002 | 1.002 | 1.002 |
| *176.gcc* | 1.335 | 1.162 | 1.068 | 1.003 | 1.003 |
| *181.mcf* | 1.489 | 1.111 | 1.005 | 1.003 | 1.003 |
| *186.crafty* | 1.249 | 1.136 | 1.039 | 1.001 | 1.001 |
| *197.parser* | 1.401 | 1.149 | 1.043 | 1.003 | 1.003 |
| *252.eon* | 1.505 | 1.231 | 1.057 | 1.008 | 1.008 |
| *253.perlbmk* | 1.613 | 1.317 | 1.209 | 1.010 | 1.020 |
| *254.gap* | 1.525 | 1.377 | 1.171 | 1.011 | 1.011 |
| *255.vortex* | 1.422 | 1.207 | 1.007 | 1.000 | 1.000 |
| *256.bzip2* | 1.098 | 1.030 | 1.001 | 1.001 | 1.001 |
| *300.twolf* | 1.217 | 1.090 | 1.005 | 1.003 | 1.003 |
| **Average** | **1.349** | **1.157** | **1.051** | **1.004** | **1.004** |

Another important statistic is the number of extra instructions generated by indirect chaining methods. Table 5-4 shows the dynamic instruction count expansion rates when the dispatch code consumes 20 instructions. It is obvious that without any register indirect jump chaining support, as in the *no_pred.no_pred* method, program performance will be unacceptable as 35% more instructions have to be executed. Conventional software prediction in *sw_pred.sw_pred* method cuts the number to about 16% by executing only the relatively short compare-and-branch code when the prediction is correct (Both shared and threaded versions result in the same instruction count expansion). Providing a dual-address RAS reduces another 10.6% of the total instructions. This is not only because return instructions now seldom reach the dispatch code, but also because the compare-and-branch code is not generated for a source return instruction. Similarly, JTLT removes almost all extra instructions for all jumps.

### 5.3.3 Branch Prediction Performance

Chaining can have a significant effect on a program's branch prediction characteristics because it can add extra control-transfer instructions or can even remove some source control transfer instructions (i.e., unconditional direct branches inside a superblock). For direct conditional branches, chaining does not change prediction performance significantly. Nonetheless, a lower number of taken branches and inlined unconditional branches tend to reduce pressure on the branch prediction hardware. On the other hand, chaining of register indirect jumps does have a large effect on branch prediction performance, and each scheme exhibits different branch prediction characteristics. Figure 5-7 shows detailed breakdown of all control transfer mispredictions that are resolved after the instruction is executed.




Sw_pred.sw_pred
No. mispredictions / 1K insns
0
2
4
6
8
10
12
14
16
18
20
164.gzip
175.vpr
176.gcc
181.mcf
186.crafty
197.parser
252.eon
253.perlbmk
254.gap
255.vortex
256.bzip2
300.twolf
A.Mean
target not found
(dispatch) indirect jump (BTB)
cond. branch (BP)
Sw_pred.sw_pred (threaded)
No. mispredictions / 1K insns
target not found
(dispatch) indirect jump (BTB)
cond. branch (BP)
Sw_pred.ras
No. mispredictions / 1K insns
target not found
return (RAS)
(dispatch) indirect jump (BTB)
cond. branch (BP)

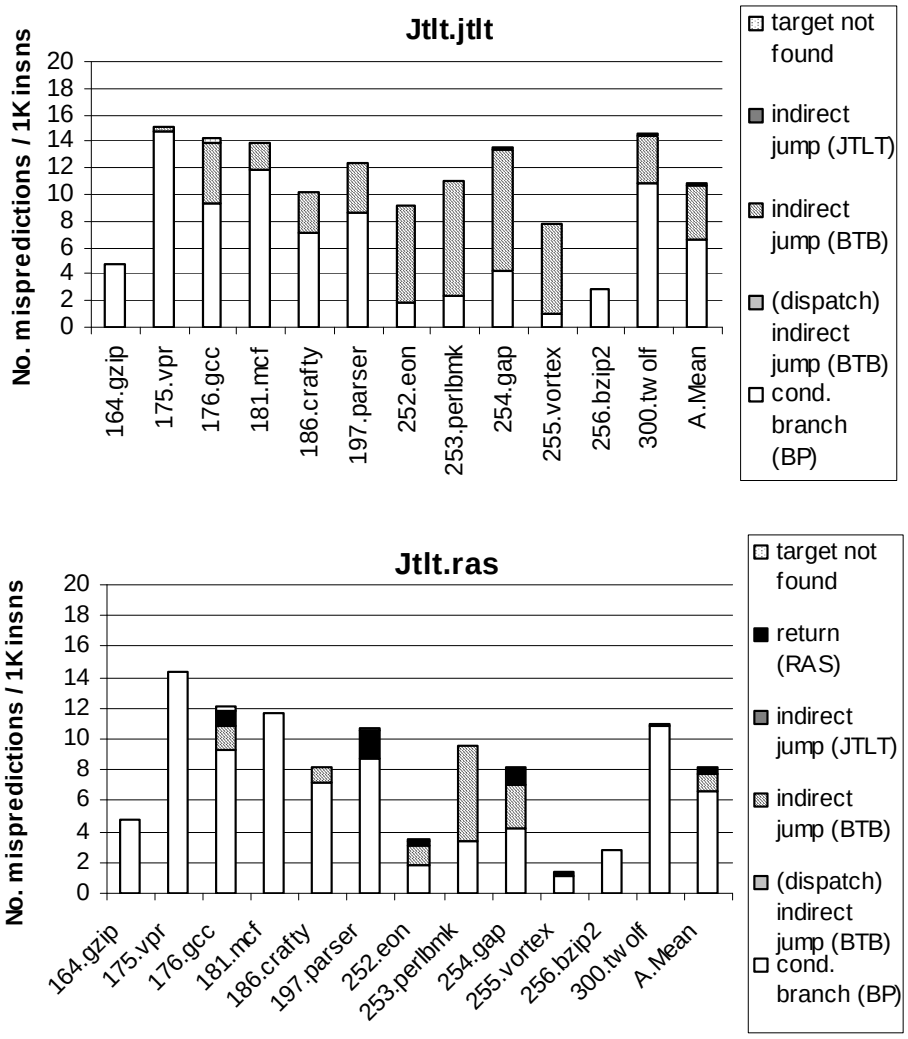



**Figure 5-7 Classification of control transfer mispredictions**

First, note that the performance impact of register-indirect jump mispredictions was not very significant (except for *253.perlbmk*, a script language interpreter) in the *original* program execution. However their effect is exacerbated in a code cache system. The conventional chaining method, *sw_pred.sw_pred*, experiences 46% more mispredictions than *original*. This increase is mostly due to the mispredictions of the indirect jump in the shared dispatch code. A threaded version, *sw_pred.sw_pred (threaded)* reduces this type of mispredictions by 44% thanks to the

private dispatch code. However it still generates 23% more mispredictions than *original*. Introducing the dual-address RAS further reduces the indirect jump misprediction to the level of *original*.

The JTLT also reduces mispredictions by cutting the dispatch code execution frequency. However, *jtlt.jtlt* still has 24% more mispredictions than the original program execution. This is about the same as the best software-based method, *sw_pred.sw_pred (threaded)*. This may seem surprising at first but it does make sense considering the prediction performance in Figure 5-5 where the BTB prediction rate is actually lower than the software prediction rate for returns.

The best method, *jtlt.ras*, has 5.6% fewer overall mispredictions than *original* due to a reduction in conditional branch mispredictions. This is possible because fewer taken branches reduce negative interference in the branch predictor pattern history table [186].

It should be pointed out that branch prediction performance comes close to the original program only after introducing the dual-address RAS. Interestingly, *sw_pred.ras* produces *fewer* mispredictions than *jtlt.jtlt*, the hardware-intensive technique.

### 5.3.4 I-Cache Performance

Another important program characteristic that can be affected by the chaining method is I-cache performance. Figure 5-8 shows that superblock-based code caching helps reduce I-cache misses, except for the threaded variant (*sw_pred.sw_pred (threaded)*) which suffers more I-cache misses due to the replicated dispatch code. In general, improved I-cache locality by superblock caching is more than enough to offset increased I-cache pressure from chaining (as is implied by the dynamic instruction count increase).

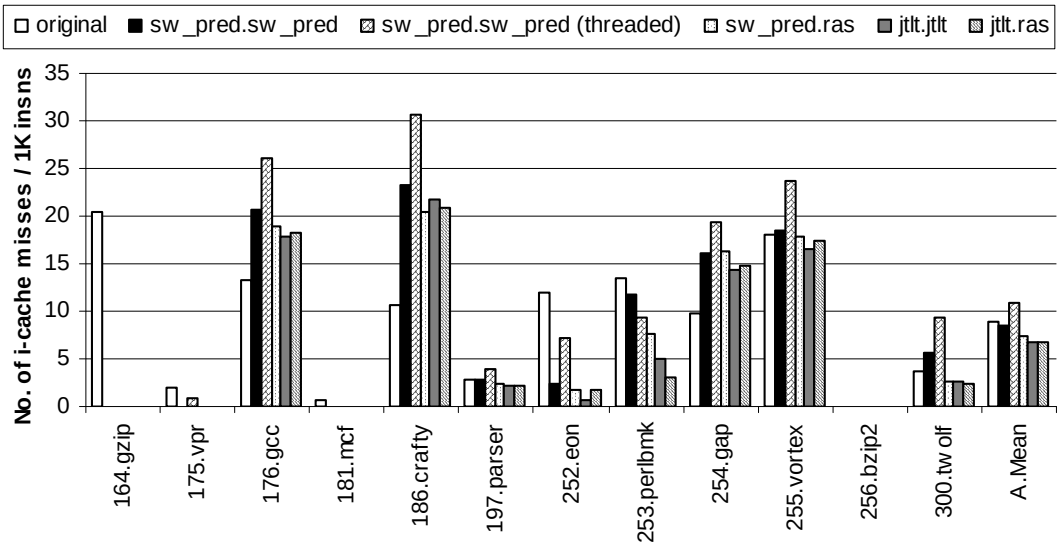



**Figure 5-8 Number of I-cache misses**

Of special interest is the dramatic miss reduction in *164.gzip*. Here, code re-layout eliminates cache thrashing. Although this is a valid optimization, it is probably limited to direct-mapped caches. If *164.gzip* is omitted, the best methods that use JTLT show a 6.3% I-cache miss reduction. (24.3% if *164.gzip* is included).

## 5.3.5 IPC Performance

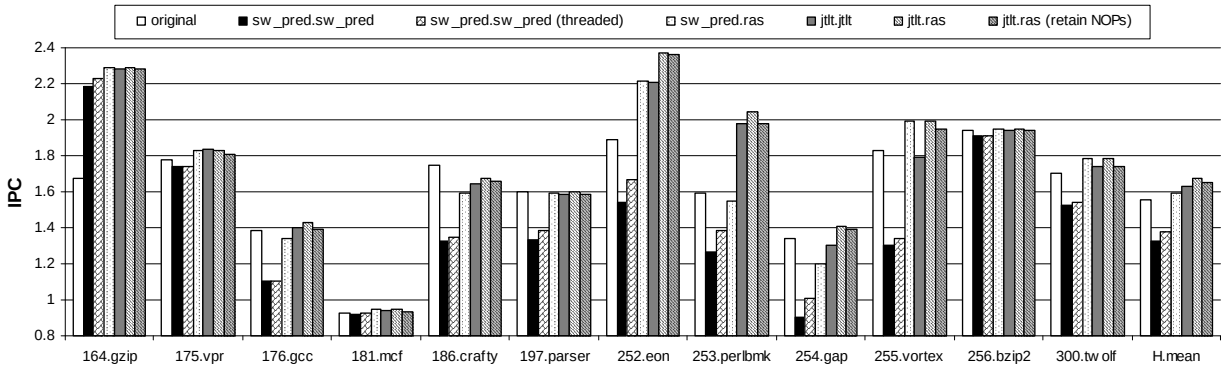



**Figure 5-9 IPC comparisons between various chaining methods**

Figure 5-9 shows overall performance in terms of the original source IPC. Note that the x axis starts from 0.8. This is to highlight the differences between chaining methods. The results show that the conventional indirect jump method that relies on software prediction (*sw_pred.sw_pred*) performs poorly, resulting in 14.6% IPC *loss*. Here, improved fetch bandwidth is offset by the chaining overhead, mostly due to increased branch mispredictions and extra instructions. Interestingly, this result contradicts previous results of HP Dynamo [15] where a 6% *speedup* is obtained just through superblock code caching. I believe the following factors contribute to the difference in results:

- Efficiency of hardware prediction mechanisms: the PA-8000 processor, used in [15], does not predict indirect jumps and always stalls fetch until the target address is resolved [107]. Hence, converting a register-indirect jump to the software prediction compare-and-branch code greatly reduces fetch stall cycles in the PA-8000. In contrast, the simulated pipeline model predicts jump target with a BTB and does not stall fetch, so a similar benefit is not realized. This is confirmed by [33]: where the Dynamo system was ported to a Pentium II platform (which does predict indirect jump targets with a BTB), resulting in substantial slowdowns due to indirect jumps – even worse than what is reported here.
- Differences in the superblock formation algorithm: the superblock construction algorithm used in this research stops constructing a superblock whenever an indirect jump is encountered. Hence some straight-line fetch optimization opportunities across highly repetitive indirect jumps are not exploited.

Returning to Figure 5-9, a threaded version, *sw_pred.sw_pred (threaded)*, performs 3.2% better than the conventional *sw_pred.sw_pred*, showing that indirect jump prediction performance improvements more than offset any losses in I-cache performance when replicated

dispatch code is used. Even this best performing software-only method still lags original program performance by 11.4%.

It is only after specialized hardware mechanisms are introduced that the identity-translation code cache system outperforms original program execution. Referring to *jtlt.jtlt*, the introduction of the JTLT greatly enhances performance both by suppressing extra instructions and in improving predictor performance. As a result, *jtlt.jtlt* achieves a 4.6% performance improvement over *original*.

However, even more important is the effect of the dual-address RAS. This is evident in the *sw_pred.ras* method; 2.1% IPC improvement is achieved *without* requiring any extra on-chip storage as in the JTLT. This is a 15.4% improvement over the best performing software-only method (and a 19.6% improvement over the conventional method, *sw_pred.sw_pred*). Finally, combining both the JTLT and dual-address RAS (*jtlt.ras*) results in a 7.7% IPC improvement over *original*.

Next consider the effect of Alpha NOP removal in superblocks. The benefit can be seen by looking at the performance of *jtlt.ras (retain NOPs)*. If not removed, these NOPs put more pressure than necessary on the fetch mechanism and reduce effective fetch bandwidth. The results show that 1.5% of the total 7.7% IPC improvement comes from NOP removal in superblocks.

### 5.3.6 Summary of Superblock Chaining Methods

From the identity translation experiment, it became obvious that the lack of accurate return prediction is one of the biggest performance limiter in code caching systems. It turned out that the dual-address return address stack is a cost-effective solution to enhance the performance of a code caching system. The jump target-address lookup table – a hardware cache of the dispatch table – also helps to further reduce the chaining overhead. On the other hand, it was shown that a dynamic

threaded code technique can be applied to improve the software-based jump prediction method when specialized hardware support is not an option.

Note that the techniques studied in this chapter can be used in many code caching systems. For the co-designed VM systems for ILDP, the full set of specialized hardware mechanisms is used. Other systems can judiciously select the most cost-effective mechanisms. Even the strictly software-based code cache systems can benefit from the dynamic threaded code technique.

## 5.4 Related Work

### 5.4.1 Profile-based Code Re-layout

Profile-guided code positioning was first introduced by Pettis and Hansen [181] to move infrequently executed code out of the main body of the program and allow a higher fraction of instructions fetched into the I-cache to be useful. This basic block arrangement also improves the accuracy of dynamic branch prediction mechanisms [186]. Ramirez, Larriba-Pey, Navarro, Torrellas, Valero [184] used the term “software trace cache” for this static compiler optimization technique which does not involve any caching *by itself*.

Regarding (hardware) trace caches [108][177][179][196], they also cache superblocks, not really “traces” in the Multiflow [149] sense. A trace cache uses hardware rather than software to form superblocks and to manage the storage. This difference results in a trade-off: trace caches do not require chaining because the hardware access mechanism is based on source PCs. However, total cache size and maximum allowable superblock size are limited by the amount of on-chip, near-processor storage. It should be noted that (hardware) trace caches and (software) superblock-based code caches are *not* mutually exclusive. For example, many of the code cache systems [26][33][34][45][232][237] run on a Pentium 4 processor [108] that employs a trace cache.

RePLay [77] can be considered an aggressive extension of the trace cache. By converting highly predictable conditional branches into assertions, rePLay increases the average superblock size to the level of software-based code cache systems for more dynamic optimization opportunities.

### 5.4.2 Superblock-based Code Cache Systems

There are a variety of systems that use code caching techniques, most of them using superblock as the basic unit within the code cache. First, there are transparent optimization systems that do not perform binary translation, but instead focus on dynamic optimizations. These systems include: Dynamo [15][33], Wiggins/Redstone [58], Mojo [45], and ROAR [159][173]. Another set of systems rely on code caching as part of a sandboxing framework for program analysis and security enhancement. These systems include DELI [62] and DynamoRIO [34]. As with the dynamic optimizers, these systems do not perform binary translation.

A second important class of systems performs binary translation from one conventional ISA to another (as well as optimization). For example, Strata [202], UQDBT [232] are designed to be retargetable to multiple target platforms. HP Shogun [143] is used to provide binary compatibility for the existing ISA programs on a platform that executes a new instruction set.

The Transmeta Crusoe Processor/Code Morphing Software [60] and IBM BOA [9] perform dynamic binary translation from an existing ISA to a proprietary ISA with performance or power efficiency (or both) as a goal. These systems use a code cache to hold translated superblocks.

Finally, high performance high-level language virtual machines (e.g., Java VM) that also use superblock-based code caching technique are emerging [26][237].

### 5.4.3 Superblock Chaining Techniques

It is widely believed that Embra machine simulator [240] first introduced the software-based translation-time jump target prediction technique. Since then most high performance dynamic optimizers [14][33][159] and binary translators [70] use a similar chaining technique.

The dual-address RAS studied in this chapter can be thought as a logical extension of the FX!32's (software-only) shadow stack mechanism [46][112], taking advantage of the co-designed VM paradigm. An IBM technology disclosure by Gschwind [94] discusses a similar idea. In this chapter, an alternative return address pair construction method utilizing JTLT was proposed as well as detailed performance studies.

A hardware cache of dispatch table entries and associated instructions have been proposed previously. In Silberman and Ebcioglu's work [207] a `SEARCH_SWITCH_TABLE` instruction queries the cache with the target SPC in a register; the target TPC is written to another register upon a hit. The next `DYNAMIC_GOTO` instruction reads the latter register and jumps to the TPC. A similar mechanism is proposed by Gschwind [93]. However, this method is undesirable in systems where no scratchpad register (for keeping the target TPC) is available to the code cache manager. It appears that the "Translation Lookaside Buffer" in Crusoe [128] is also a hardware cache of dispatch table entries. It is not clear, however, exactly how the mechanism is used from the publicly available patent document.

ROAR from University of Illinois [159] is an interesting hybrid system. Although the code cache is placed in memory, the transition between native execution and interpretation is automatically performed by hardware. When an optimized superblock is put into the code cache, its starting TPC is explicitly written into the branch target buffer by the optimizer software. The next time a corresponding branch instruction is fetched, control is transferred to the optimized superblock by the BTB. This system also uses the software jump target prediction technique;

however when the software prediction fails, program control falls back to the source indirect jump instruction. Therefore no dispatch table lookup is performed. Sooner or later, the program will again reach an optimized superblock. A limitation of this approach is that its application is restricted to dynamic optimization. Many dynamic binary translation systems, such as co-designed virtual machines and high level programming language virtual machines, *cannot* use a hardware interpreter (for executing out-of-code-cache instructions efficiently).

# Chapter 6 Experimental Framework

An accurate experimental framework that faithfully models the system under study is one of the most important parts of computer architecture research. It is especially important for the type of study in this thesis because, by definition, the research encompasses many different aspects of computing, e.g., instruction sets, microarchitectures, and binary translation, at the same time. This requirement led to a development of a set of self-complete simulation infrastructures, much of them built from scratch.

This chapter describes the overall experimental framework used in the thesis. First, requirements of the simulation infrastructure are identified and the overall experimental methodology is described. Next, the chosen pipeline model – largely modeled after IBM POWER4 – and associated design trade-offs and speculation mechanisms are described. The ILDP virtual machine system simulation framework builds on this baseline simulator by adding a dynamic binary translation mechanism and changing the processor pipeline to properly model the ILDP microarchitecture. Lastly, related publicly available research infrastructures are described.

## 6.1 Objective

Modeling the entire co-designed virtual machine in great detail is a substantial engineering task that typically requires many person-years and is beyond the scope of the thesis research. Instead, the focus of the simulation infrastructure was set to faithful modeling of the baseline and ILDP microarchitectures and the dynamic binary translation mechanism.

### 6.1.1 Limits of the Previous Research Simulators

The initial comparative experimental studies of the accumulator-oriented ILDP system against conventional superscalar processors [133][134] were performed on a modified version of SimpleScalar 3.0C toolset [35]. Although it is true that the toolset, `sim-outorder` in particular, provided a convenient platform to develop a useful research infrastructure, its fundamental limitation – being an *idealized superscalar execution model* simulator, not a *realistic pipeline* simulator – prevented fair performance and complexity comparisons. Some of the limits of the toolset are listed below in descending order of importance to the thesis research.

- **Lack of adequate memory subsystem modeling**: The biggest source of error comes from inadequate modeling of implementation details and resource limits on caches and various memory-related buffers. The same observation is made by Perez, Mouchard, and Temam [180].
- **Simplistic pipeline structures**: `Sim-outorder`'s simplistic 5-stage pipeline model is not adequate for typical current-generation high-performance superscalar processor designs.[19]
- **Separation of program execution and timing simulation**: This can be considered as an advantage in the sense that it allows quicker development of a timing simulator. However, this approach is vulnerable to obscure timing bugs as there is no clear way to verify if the

[19] A common misconception is that a longer pipeline can be adequately modeled by simply using larger branch misprediction penalties. This is not completely true because a longer pipeline inevitably introduces underutilization of pipeline resources. For example, fetch "bubbles" – wasted cycles to redirect fetch for a predicted taken branch – are one of the well-known inefficiencies for longer pipelines that cannot be modeled with a larger misprediction penalty alone.

modeled timing behavior is correct [39][180][208]. Integrating correctness requirements into the timing simulator helps to build a *precise* model [39].

- **Idealized execution model**: Although the out-of-order superscalar execution model itself (based on the Register Update Unit (RUU) mechanism [219]) used in `sim-outorder` is a fine one, most current designs make certain complexity-effective trade-offs in one form or another, mostly to meet the given timing requirements.

All in all, these limitations lead to unrealistically high performance estimates for a baseline superscalar processor, giving unfair disadvantage to the ILDP system that must model complexity-effective trade-offs, by definition. To circumvent these limitations, a complete re-write of the simulation infrastructure was called for.

## 6.2 Simulation Framework

### 6.2.1 Overall Simulation Methodology

To meet the goals described in the previous section, it was decided early on that the DBT functionality should be built into, and tightly integrated with, a microarchitecture timing simulator[20]. Although a translating virtual machine system typically goes through many operating mode changes (interpretation, translation, trap handling, etc.), only the *native* execution of the translated code (which accounts for more than 99% of the total instructions as will be shown in section 7.3.2), is timing-simulated. Other operating modes are modeled at the function level. Nonetheless their

[20] Obviously the most accurate way to measure the translation overhead is to run the DBT code, statically-compiled for the ILDP ISA, on the ILDP timing simulator and measure the number of cycles to translate collected superblocks. However, it is not very practical to construct a production-quality compiler that can compile large programs – another substantial engineering task – just for this purpose.

effects, e.g., dynamic translation overheads, are taken into account by applying an estimated average number of cycles per virtual ISA instruction to the total execution time. This is done by first compiling the whole source code tree of the DBT/timing simulator hybrid framework for the Alpha ISA, then running benchmarks on the framework as usual and measuring the translation overhead as the average number of dynamically executed Alpha ISA instructions for translating a single source Alpha ISA instruction. Assuming that the translator code will have similar IPC performance as translated codes, this gives a reasonably good estimate of the overhead, albeit indirectly. This is the same method use in the evaluation of IBM DAISY [70]. Modeling the operating modes is explained further in section 6.4.1.1.

Other aspects of a co-designed virtual machine system, memory management below the operating system for example, have an impact on overall performance. However, these issues are largely orthogonal to the focus of the thesis. Besides, it is not very practical to model all the issues in a research simulator mostly built from scratch (other than system call emulators and a program loader). Instead, the thesis research focuses on evaluating the *relative* performance differences between conventional superscalar designs and the ILDP VM system. More specifically, the system-level activities such as program loading and system calls are emulated at a functional level.

### 6.2.2 Modeling Microarchitectures

The following principles were established in building the detailed microarchitecture simulator:

- **Combine program execution and timing simulation**: Maintain data values in all pipeline structures, e.g., caches, buffers, and bypasses, as well as physical register files. Instructions flowing within the pipeline capture these data values, for example, from bypasses. When an instruction is retired, important architected state values associated with the instruction, e.g.,

the program counter, result data, dynamically calculated memory addresses, etc., are compared against their counterparts in a purely functional simulator. A mismatch reveals a timing modeling error (sometimes complex interaction of multiple timing errors). That is, even an obscure timing "bug" that could have gone unnoticed in a traditional simulator is detected as part of the simulation process.

- **Model a realistic cache subsystem**: The cache hit/miss probe is separated from its timing behavior. There are many crucial elements such as resource contention that affect effective cache latency. Constructing a timing model *outside* the cache modules using finite-size buffers, such as load miss queue, allows accurate and flexible modeling of cache timing behavior. Caches also contain actual data values, enforcing correct timing modeling between closely related buffers.
- **Model a realistic, complexity-effective pipeline**: Current generation high performance superscalar designs are surveyed to arrive at a representative pipeline. Important implementation details such as finite number of register ports and operand bypass network lanes are modeled. This is important not just for performance evaluation but also for complexity estimates.

In short, the above requirements *force* construction of a microarchitecture simulator that will provide better reference *and* base platform to develop a detailed ILDP microarchitecture simulator.

### 6.2.3 Modeling Dynamic Binary Translation

In line with the microarchitecture simulator construction approach, the following principles were established:

- **Implement the DBT algorithm efficiently**: One of the key points of this particular style of a co-designed VM system is that its binary translation algorithm is simpler and hence faster, compared to previous systems based on VLIW ISAs. Therefore it is important to build a fast *and* fully-working DBT mechanism for measuring the translation overheads accurately.
- **Generate actual ILDP ISA translations**: Using an immediate representation (IR) format allows fast development of a flexible translator. A downside of this approach is that it is relatively easy to introduce (mostly likely unrealistically generous) evaluation errors. To avoid this pitfall, a concrete, finite-bit-width ISA format is employed. Combined with the principle of maintaining actual instruction bit values within the simulator, this helps to enforce correct performance and complexity evaluation.

## 6.3 Baseline Superscalar Model

This is the first step in the development of the entire research infrastructure. Therefore, the baseline model was thoroughly examined before the ILDP simulator was built on top of it. Also this step is important for checking out various replay mechanisms and complexity-effective design trade-offs.

### 6.3.1 Choosing a Baseline Model: IBM POWER4-like Pipeline

Building a baseline model starts with choosing a well-designed conventional processor. To this end, I collected and studied publicly available design documents of recent out-of-order superscalar processors including the MIPS R10000 [241], DEC Alpha 21264 [131], Intel Pentium

Pro [98], Pentium 4 [28][108][152], Pentium M [90], AMD Athlon [63][64], Opteron [129], Fujitsu SPARC64 V [198], and IBM POWER4 [228]. Of these existing designs, IBM POWER4 was chosen for the following reasons:

- **Modern, complexity-effective design**:
- **Detailed, publicly available documentation**: Many of its complexity-effective design trade-offs are well documented in depth [228].
- **PowerPC RISC ISA**: Even though PowerPC ISA is fairly complex for a RISC ISA [214], it still is closer to Alpha EV6, the virtual ISA in the research, compared with other complex ISAs such as the Intel x86.

It is important to note that the baseline pipeline is *largely* modeled after POWER4, and is not intended to be an exact replica. That is, there are certain differences, mostly due to using a different ISA (Alpha).[21] In general, the baseline pipeline should be considered slightly optimistic – or forgiving – as compared to real-world designs that have even more strict implementation limits.

### 6.3.2 Pipeline Overview

Figure 6-1 shows a high-level pipeline diagram of the baseline superscalar processor. As with Figure 3-1, the boxes represent pipeline stages rather than actual hardware structures. Thick lines represent instructions flowing through the pipeline; these lines as pipeline lanes. Various replay and cache miss paths are shown with dotted lines.

---

[21] For example, unlike POWER4, the baseline pipeline does not have a separate (in-order) issue queue for control transfer instructions. PowerPC ISA uses separate registers for these instructions, which in turn enables a simple, separate issue path – a complexity-effective design choice.

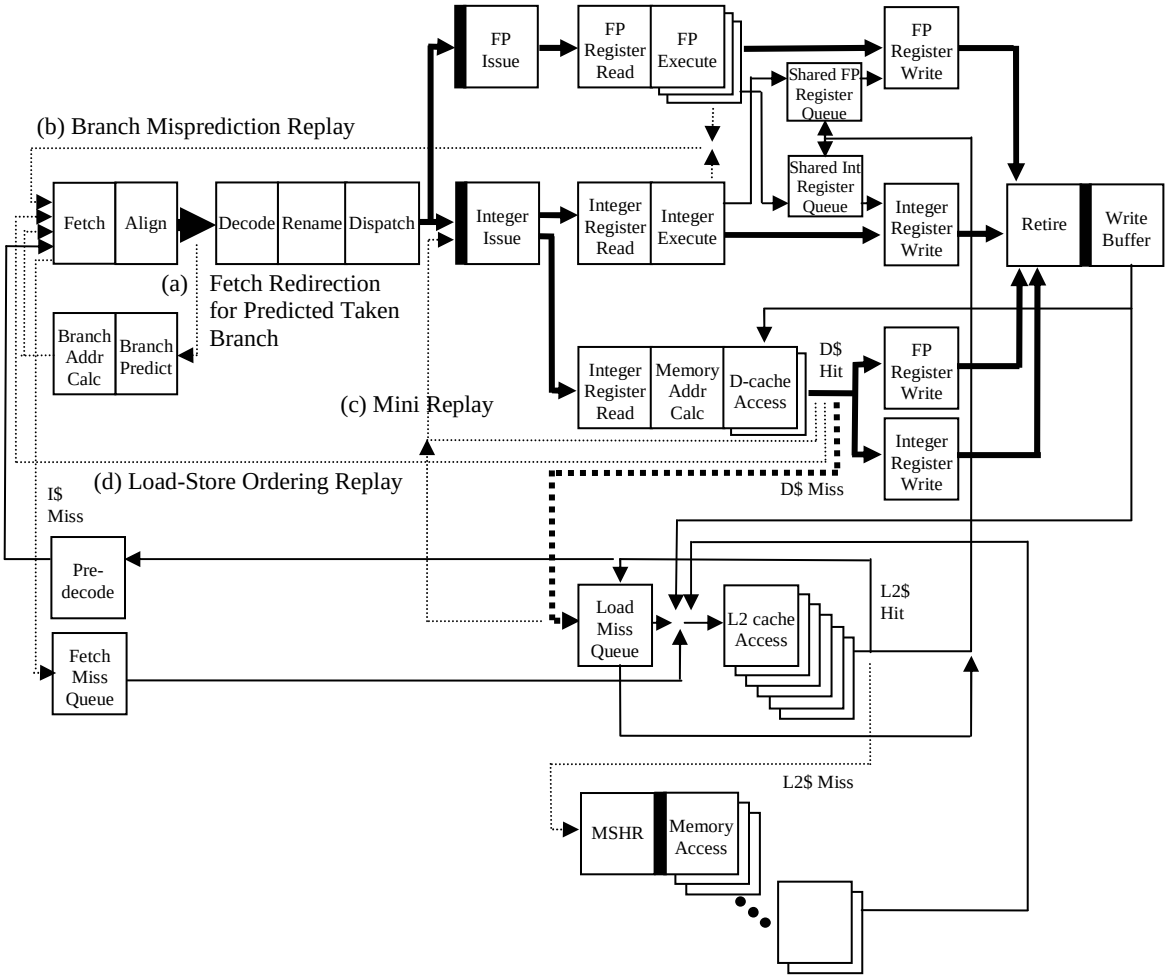



**Figure 6-1 High-level block diagram of the baseline pipeline**

Instructions are first fetched from an L1 I-cache and scanned for possible control transfers as they are aligned to maintain the correct fetch order. If a control transfer instruction is found, branch predictors are consulted, based on an instruction's associated predecode information. For a predicted taken conditional branch, the target address is calculated from the PC and immediate field values. The target address of an indirect jump is obtained from a return stack (for return instructions) or a branch target buffer (for non-return jump instructions). Alpha ISA NOPs are properly recognized and removed in the decode stage.

The pipeline uses separate integer and floating-point physical register files for register renaming; naturally, there are separate register rename maps and associated scoreboards for each operand type. A conditional move instruction – the only instruction type that requires three input operands – is decomposed into two micro-instructions in the same way as the Alpha 21264 processor [131]. After register operands are renamed, each instruction is assigned a reorder buffer entry in the dispatch stage where it is sent to an issue queue based on its source operand type. Memory instructions are assigned either a load queue or a store queue entry. Precise state maintenance [211] is achieved through the combination of rename maps, reorder buffer, store queue, and load queue; all are allocated and de-allocated in program order.

Instructions are issued out-of-order from the appropriate issue queue. Memory instructions are issued from the integer issue queue because their operand for (address) *calculation*, an address register, is always an integer. Note that memory instructions are issued based on the availability of their address operand alone, and are not constrained by the address resolution of other logically-older memory instructions.

All issued instructions get their operand values from the physical register file or bypass network. One exception is store data operands, which are read by the associated store entry *within* the store queue, *independent* of store address resolution. All instructions that produce a same-type result value as source operand(s) are given two register read ports and a write port – i.e., a lane per instruction. Similarly, one register read and two write ports (one for possible integer data and another for possible floating-point data) are reserved for each load instruction lane. An instruction that reads an integer operand and writes a floating-point result or vice versa is allocated an entry in a shared register write queue (SRWQ).

After its effective address is calculated, a load instruction probes both the L1 D-cache and the STQ for possible forwarding. If neither was successful, the missed load is allocated a combining

load miss queue entry. When a missed load in the LMQ receives its data, it is put into one of the shared register write queues. Other than that, the memory subsystem including the L2 cache is largely the same as the one used in the ILDP microarchitecture.

The system bus interface is the least accurately modeled part of the simulator, mostly due to the lack of a detailed description of the bus timing behavior. However, given the relatively low L2 cache miss rates found in the SPEC CPU2000 integer benchmarks that are used in the evaluation, this effect should be small. Nonetheless there are small number of benchmarks that do exhibit large fraction of L2 cache misses. Slightly aggressive memory latencies are used to make up for the simple memory bus model.

The speculative nature of the memory instruction issue can sometimes lead to replay conditions. Note that prefetches (load instructions into a constant zero register) are properly recognized; when a prefetch lead to a replay condition, it is simply marked complete and replay is not performed.

### 6.3.3 Complexity-Effective Design Trade-Offs

The following design trade-offs used in the baseline model are frequently found in modern designs:

- **Separate hardware structures for different operand types**: Most instructions use same type of data for both operands and results. Exploiting this program characteristic allows relatively smaller hardware structures for the register rename mappers, scoreboards, issue queues, physical register files, and operand bypass networks. Shared buffers such as SRWQs take care of the small number of instructions that generate communications between two largely independent structures, e.g., integer and floating-point register files.

- **Limited numbers of ports and bypasses**: For SRAM-array-based structures such as register file and caches, latency is linearly proportional to the number of ports [175]. In general, limiting the number of ports of a given structure reduces the maximum attainable parallelism. Nonetheless, it is often beneficial to reduce the number of ports, hence reducing the number of cycles to access that structure, to avoid increasing pipeline inefficiencies. A well-known example is in physical register file; it is possible to design a multi-cycle register file with full internal bypasses, but doing so increases the number of shadow cycles (shown in Figure 3-3) resulting in increased energy waste [138][164]. Limiting the number of ports and bypasses naturally leads to reduction in issue width. Section 7.2 shows that the resulting performance loss is generally small enough to encourage this trade-off.
- **Simplified issue logic**: Modern out-of-order superscalar processors employ segmented issue logic in one form or another to meet clock frequency goals [63][98][108][131][198][228]. It is shown in the next chapter that the effect of segmented issue logic is minimal, especially in the wake of other events such as L2 cache misses that have an order of magnitude greater impact on overall performance. Figure 6-2 shows a 4-way segmented integer issue queue along with operand bypass network used in the baseline model. Note the large fan-out, i.e., capacitive loading of the bypass network.

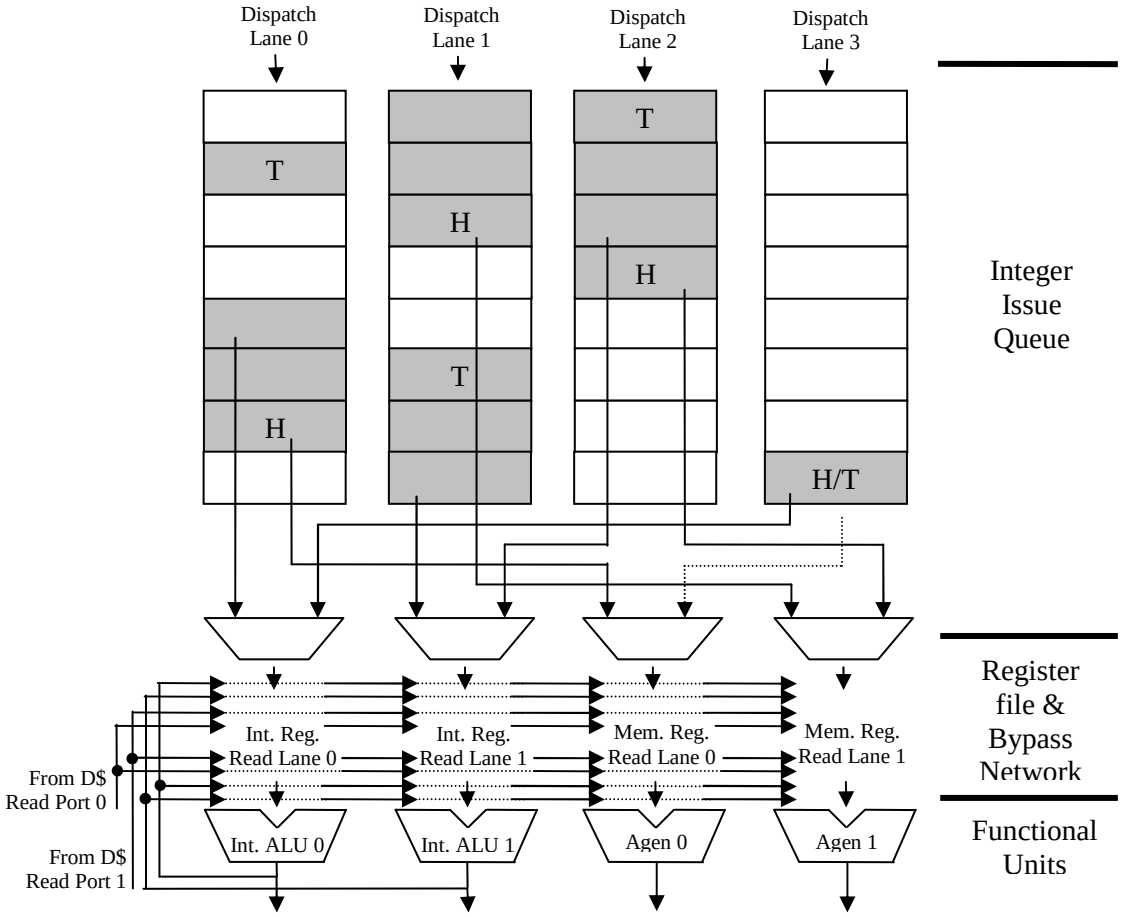



**Figure 6-2 Segmented issue queue and bypass network used in baseline model**

- **Separate LDQ from STQ**: Making data forwarding from an older store to a younger load at least as fast as an L1 D-cache hit latency substantially simplifies pipeline design [7][242]. Separating LDQ from STQ is a common way to reduce the effective size of the dynamic memory disambiguation mechanism [98][108][131][198][228]. The downside is that now there are two places that can generate memory ordering violation replays.
- **Reliance on replays**: In some cases, checking all conflict conditions before accessing a structure may complicate the design or even render it impossible. Using speculative accesses and relying on replays can sometimes provide a simpler overall solution. This is

especially important for memory-related structures that have uncertainties by definition [242]. For example, it would not make much sense to have the issue queue check for a possible LMQ full condition beforehand. The baseline model uses a simple PC-based dependence predictor [165] (same as the one used in the ILDP microarchitecture) in conjunction with speculative memory instruction issue.

## 6.4 ILDP System Model

### 6.4.1 Modeling Dynamic Binary Translation

The next step after modeling the baseline pipeline was to build an ILDP dynamic binary translation mechanism. To meet this goal, the DBT mechanism (for interpretation and translation modes) was first developed and combined with an ILDP ISA functional simulator (for native execution mode) resulting in a self-complete ILDP VM system model. A timing-accurate ILDP pipeline model was plugged in to the framework after the functionality of the DBT was fully verified.

#### 6.4.1.1 Framework Mode Changes

```
While (TRUE)
{
  if (cur_mode == MODE_NATIVE)
    cur_mode = mode_timing_execute_natively();
  else if (cur_mode == MODE_INTERPRET)
    cur_mode = mode_interpret();
  else if (cur_mode == MODE_TRANSLATE)
    cur_mode = mode_translate();
}
```

(a) Upper-most level loop

```
int mode_timing_execute_natively()
{
  ildp_pipe_reset();
  for (;;)
  {
    /* model L2 cache/memory */

    next_mode = ildp_retire();
    if (next_mode != MODE_NO_CHANGE)
      return next_mode;

    /* model the rest of the pipe */

    sim_cycle++;
  }
}
```

(b) Pipeline simulator top-level loop

**Figure 6-3 Top-level simulator loop showing operating modes**

Figure 6-3 shows how the operating modes are integrated with a timing simulator at the highest level of the simulation framework. Note that the pipeline is flushed between mode changes.

#### 6.4.1.2 Dispatch Table Lookup Mechanism

Even with the specialized hardware support mechanisms discussed in Chapter 5, a control transfer instruction can sometimes fall through to the ultimate safety-net mechanism commonly called a dispatch table lookup. This is essentially a hash table search with a source target address; if there is a match, the corresponding target translation's start address is returned. Care must be taken to achieve the highest possible lookup speed to avoid unacceptable slowdowns in certain pathological cases.

```
/* dt_hash_map is randomized at start to minimize conflicts */

/* generate hash key from the source PC */
hash_key = dt_hash_map[(src_PC >> 2) & MAX_XC_MASK];
dispatch_table_entry = dt[hash_key];

while (dispatch_table_entry != NULL)
{
  if (dispatch_table_entry->src_PC == src_PC)
    return dispatch_table_entry->xl_PC;

  dispatch_table_entry = dispatch_table_entry->next;
}

/* not found */
return 0;
```

**Figure 6-4 Dispatch table lookup algorithm**

To accurately model the dispatch table lookup behavior, the dispatch lookup algorithm in Figure 6-4 was first written and hand-optimized in Alpha assembly language, and then converted to the ILDP ISA. The resulting lookup code consists of total of 18 Alpha instructions and 22 ILDP instructions, respectively. An x86 version of HP Dynamo dynamic optimizer [33] reported 15 *x86* instructions for the same function.

The dispatch table, the translation cache, and other support data structures in the hidden memory are shown in Figure 6-5.

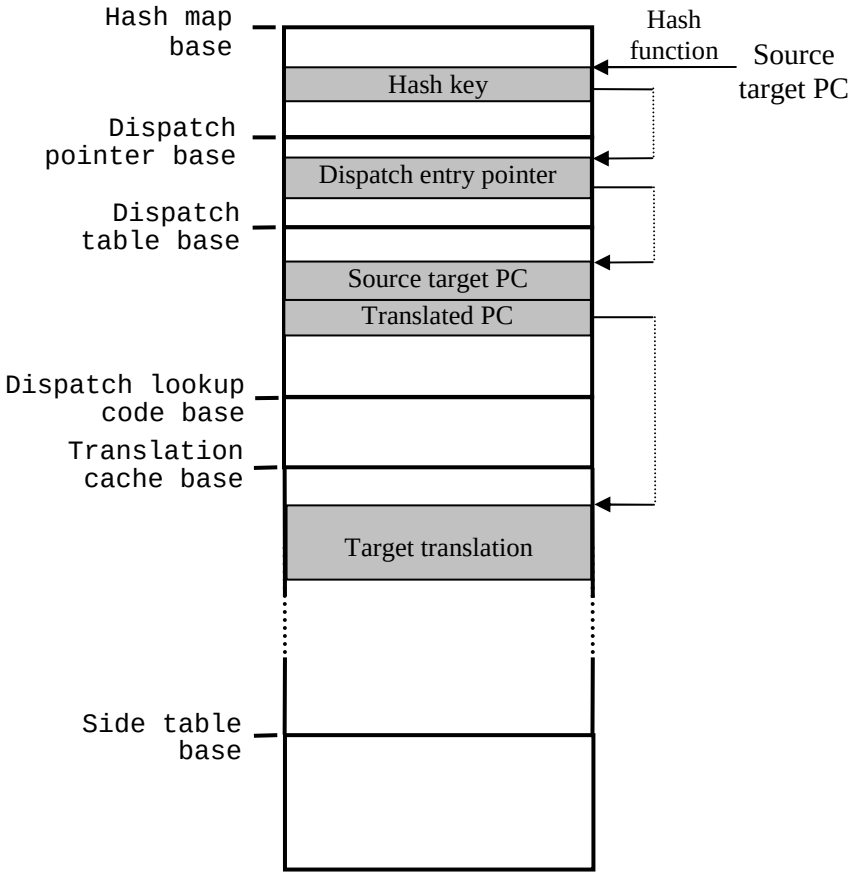



**Figure 6-5 Map of the hidden memory area**

#### 6.4.1.3 Effect of Dynamic Binary Translation on Caches and Predictors

In general, the timing performance of the DBT is separately evaluated using the estimated average number of cycles per V-ISA instruction. In reality, the DBT actions affect the performance of the native execution mode, too, mostly through cache and predictor interference. For example, dynamically translated code is written to the translation cache as data and in turn, ends up being cached in the L2 cache, possibly replacing other data. More importantly, if there is a previously generated translation that happens to share an I-cache line with the newly created one, that I-cache line has to be invalidated for correct program execution (then cached again when needed later).

These effects are modeled correctly; otherwise the simulation framework that maintains data values within caches will fail.

The effect of DBT actions on predictors is not as strongly enforced as caches, though. Predictors are performance *features* not correctness *requirements*. Nonetheless, the hardware dual-address return stack is properly maintained by the DBT. For example, when a function call instruction is emulated in interpretation or translation mode, the return stack is explicitly pushed by the DBT. In general, updating other predictors to reflect the DBT action is not possible.

### 6.4.2 Modeling ILDP Pipeline

The baseline pipeline simulator is heavily modified to properly model the ILDP pipeline. The most substantial changes are made in the following areas: instruction scanning/alignment, accumulator renaming and steering, overall out-of-order execution model, micro-instruction (such as dynamically generated effective address calculation) handling.

There is one complication from supporting the framework mode change – when a mode change is needed (by a retiring trap-to-interpreter instruction for example), there is a chance that some cache transfers may still be in flight. As a result, the pipeline can only be drained after all the pending transactions are finished (and leave the caches in correct state). This adds extra cycles to switching between framework modes.

## 6.5 Evaluation Criteria

The goal of this research is to show that the ILDP paradigm is capable of performing better than conventional superscalar designs while substantially reducing complexity in key pipeline hardware structures. The reduction in hardware complexity can be exploited in two ways; higher clock frequency for better performance or reduced power consumption while still achieving the

same level of clock frequency. Or it may be possible to find a better balance between high performance and low power consumption than conventional design approaches can provide. Therefore, it is not enough to evaluate only the performance aspect of the ILDP paradigm; the simplicity advantages of the ILDP system should also be evaluated.

### 6.5.1 Performance

In general, performance of a computer system can be modeled with the combination of the following three factors [212]:

Execution time = number of instructions * clock cycles per instruction (CPI) * clock cycle time

= number of instructions / (IPC * clock frequency)

**Figure 6-6 The three components of computer performance**

Of the three components, typical microarchitecture research concentrates on improving IPC performance by introducing new microarchitecture techniques. Recently research on improving clock frequency, has become popular. Most of these microarchitecture research deals with IPC and sometimes clock frequency estimates.

Realistically estimating achievable clock frequency requires a reasonably low-level physical design of the entire pipeline using the actual technology parameters. The same is true for estimating power consumption of a given design. For these reasons, the thesis research does not try to provide an estimated clock frequency in picoseconds or a power consumption level in watts. Instead, the thesis concentrates on the instructions per cycle performance and comparative complexity estimates of ILDP and conventional superscalar pipelines.

Unlike traditional microarchitecture studies, these two factors alone are not sufficient for the thesis research; the change in the number of dynamic instructions resulting from binary

translation should also be accounted for. In a co-designed virtual machine system that must maintain strict architectural compatibility, dynamic binary translation often leads to code expansion. This is partly because the translator can only generate instructions at the same or finer architectural granularity than the source ISA. The proposed hardware-software co-designed support mechanisms for suppressing code expansion are evaluated in section 7.3.2.

### 6.5.2 Simplicity

Admittedly, the simplicity advantage of the ILDP system is harder to quantify compared to performance. The reduction of complexities in key hardware structures is summarized in a table in section 7.3.1. The thesis notes that the ILDP microarchitecture provides a unique combination of physical register file based register renaming model and a capture-before-issue operand capture model. This new operand capture model leads to a dramatic reduction in mini replays associated with physical register file based designs while reducing reliance on bypass networks typical in the machines that capture operands before instruction issue. As a result, a simpler design with less low-level speculation techniques, such as load latency speculation, is possible. The reduction of mini replays is shown in section 7.3.3.3.

Another type of complexity reduction comes from simpler translation algorithms compared with the VLIW-based ones used in previous co-designed VM systems. Measurement of average number of dynamic instructions in the translator is used as a way to quantify the complexity of the translation mechanisms.

## 6.6 Related Work

### 6.6.1 Microarchitecture Simulators

By far the most popular simulator within the computer architecture research community is the `sim-outorder` timing simulator from the SimpleScalar toolset [35]. Since its introduction in 1997, many researchers used the simulator as a baseline platform to apply and evaluate new microarchitecture techniques, almost to the point of abuse. To summarize, the problem is that many researchers use this simplistic *execution model* simulator to evaluate microarchitecture ideas that only make sense within the context of today's deeply pipelined designs with complex cache and memory subsystems.

There have been many efforts to correct this situation. `Sim-alpha` [61] was a detailed-up version of `sim-outorder`; it was validated against an Alpha 21264 system, and it was shown that a simplistic simulator can lead to wrong conclusions. Two issues prevented `sim-alpha` from being used in this research. First, it still separates timing simulation from program execution. Second, a 7-stage Alpha 21264 pipeline was not a current-generation design by the time the simulation infrastructure for this research was constructed. It is believed that the Multiscalar simulator [220] is the first research microarchitecture simulator that combines program execution and timing modeling. `Pharmsim` [39], an evolution from `sim-OS` system simulator [194], also does that and goes even further to simulate the whole system including OS actions and I/O device timing. The latter worked as a disadvantage, however, to integrate a dynamic binary translation mechanism in a simple and efficient manner.

Recently, two simulators based on `MicroLib` [180] that combine program execution and timing simulation were made public. On the industry side, a production-level performance simulator

for Intel Pentium 4 processor [208] uses the same verification technique used by the simulators built for the this thesis research. I believe the detail level of the baseline simulator is roughly equivalent to the IBM research's `Turandot` [114], a trace-driven timing simulator that more or less models the IBM POWER4 pipeline.

### 6.6.2 Code Cache Frameworks

A simulator performs many of the same functions as a virtual machine. A common way to speed up a virtual machine is to translate and cache most frequently executed code for native execution. Hence, most dynamic translation/optimization systems have a similar code caching framework that switches modes between interpretation, translation (and/or optimization), and native execution. These related systems are described in section 4.4.

Although there are many code cache systems in both academia and industry, only a few are publicly available. The DBT framework in the thesis uses a slightly modified superblock construction algorithm of Dynamo described in [14]. One important research infrastructure that is publicly available is the DAISY simulator from the IBM research [119]. Although it is a sufficiently self-complete platform that allows further development, it became apparent that building a simple dynamic translation mechanism on top of a familiar timing simulator would be much simpler than modifying the VLIW-based DAISY timing simulator to model the ILDP microarchitecture.

# Chapter 7 Evaluation

This chapter presents empirical evaluation of the ILDP system. First, the benchmark programs and other related simulation setup are described. The baseline superscalar model is thoroughly evaluated and key insights from the experiments are presented. The evaluation of the ILDP virtual machine system consists of the following: dynamic binary translation characteristics and translation overhead evaluation, the IPC performance comparison against the baseline superscalar model and performance variation over machine parameter variations, and complexity comparisons of the important pipeline hardware structures in the baseline and ILDP pipelines.

## 7.1 Simulation Setup

To collect statistics, I use the SPEC CPU2000 integer benchmarks [105] compiled for the Alpha EV6 ISA at the `base` optimization level (`–arch ev6 –non_shared –fast`). Note that the `–fast` option includes aggressive instruction scheduling, procedure inlining, and loop unrolling. The compiler flags are the same as those reported for Compaq AlphaServer ES40 SPEC CPU2000 submission results. DEC C++ V.6.1-027 (for *252.eon*) and C V.5.9-005 (for the rest) compilers were used on Digital UNIX 4.0-1229. Of the three input data sets, the smallest `test` sets are used. This choice was made for the purpose of covering all program phases. Skipping the initialization phase and simulating only part of program using the `ref` input sets, as is frequently done in microarchitecture research, would likely have exaggerated the benefits of profile-based dynamic binary translation. All benchmarks were run to completion or 4 billion instructions. For those benchmarks that finish before 1 billion instructions, the `train` input sets were used.

For translation, a maximum superblock size of 200 and a counter threshold of 50 are used. A smaller superblock size of 50 was also tried but it turned out that this size was not large enough to

obtain performance benefits from code re-layout. This observation is in line with the HP Dynamo report [14]. By default, eight logical accumulators are used. Four and six accumulator results are also reported.

In this evaluation, an unlimited number of counters for superblock start candidates are used to simplify the simulation framework design. For programs the size of the SPEC benchmarks, however, the number of counters is relatively small. Also, the small static code size of the SPEC benchmarks (all are less than 1 Mbytes except *176.gcc* at 2.3 Mbytes) means that they would comfortably fit into a reasonably sized translation cache. Thus translation cache management is not required; however other research indicates that this overhead is generally negligible [14]. In fact, there may be a performance *cost* in not occasionally flushing translation cache entries. Some translated superblocks may be sub-optimal because once a translation is put into the translation cache there is no second chance for forming a different translation starting from the same address in the current DBT implementation. In Dynamo, for example, the code cache is flushed when there is a change of program phase (indicated by abrupt increase of superblock generation rate) thereby evicting infrequent superblocks from the cache and allowing new superblock formation.

## 7.2 Validation of Baseline Model

The evaluation began with the validation of the baseline superscalar simulator. This is important in two ways. First, the ILDP system model is built on top of the baseline simulator. Therefore the baseline model should be validated before the experiments of the ILDP system are conducted. Secondly, a close investigation of performance bottlenecks in a realistic pipeline model gives valuable insights in pipeline design trade-offs.

### 7.2.1 Idealized Performance Evaluation

By construction, it is very difficult for the timing simulators in the thesis to *overestimate* performance of the given pipeline design. For example, if some of the bypass timing is modeled too aggressively, instructions will eventually capture incorrect values. Sooner or later, this error is uncovered by the "golden" simulator when the first instruction with the erroneous data value retires. However, it is possible that unrealistically *pessimistic* timing models can go unnoticed. Hence the goal of the initial validation was set to prevent underestimation of performance.

To achieve this goal, the baseline pipeline was initially simulated with perfect predictors and caches. Ideally, this will lead to the estimation of the sustained performance under ideal conditions (i.e., no miss and replay events). This steady-state performance should approximate the overall machine width. Once this expected ideal performance level is confirmed, the effects of transient performance penalties can be applied by introducing non-ideal predictors, caches, and memory timing models. This methodology is in line with the one used by Karkhanis and Smith [127].

Figure 7-1 shows the idealized baseline pipeline model. Perfect branch prediction is implemented with a second functional simulator that provides branch outcomes early in the pipeline. This simulator was placed in such a way that avoids *any* fetch redirection bubble cycles. This is to isolate the effect of discontinuity in the fetch stream created by the taken branches. The functional simulator also provides memory addresses for perfect disambiguation.

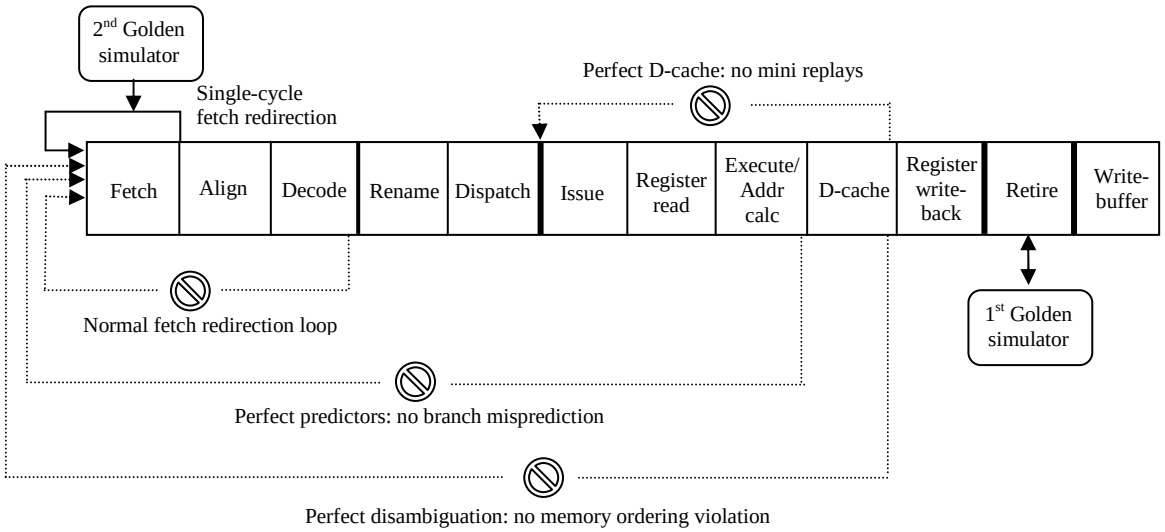



**Figure 7-1 Ideal baseline pipeline**

**Table 7-1 Machine configurations used in idealized performance evaluation**

| | Idealized wide machines | Baseline machine (wider issue) |
|---|---|---|
| Fetch bandwidth | 8 | |
| Decode/rename/dispatch bandwidth | 4 | |
| Branch/store/other retire bandwidth | 4/4/4 | 1/1/4 |
| Fetch redirection latency | 0 | 2 |
| Issue window | 512, 128, 32 | 32 |
| Issue logic | Fully shared, unified | 4 x 8 as in Figure 6-2 |
| Issue width | 4 integer, 4 memory, 4 floating-point | |
| Reorder buffer | 512, 128, 128 | 128 |
| Store queue/load queue | 512/512, 128/128, 128/128 | 32/48 |
| Physical register file | 512/512, 128/128, 128/128 | 80/80 |

To isolate the effects of the pipeline implementation limits as much as possible, the machine parameters are set to be sufficiently larger than a typical 4-way superscalar pipeline configuration. In Table 7-1, the artificially large machine parameters are shown side-by-side to the corresponding baseline parameters. Note that the issue width, 4 for each functional unit type, in the right-hand side is still much larger than the actual configuration (2 for each functional unit type) used throughout the evaluation. This is intended to isolate the effect of the number of functional units from the idealized performance evaluation. The complete set of parameters for the realistic pipeline is shown side-by-side with the corresponding ILDP parameters in section 7.3.1.

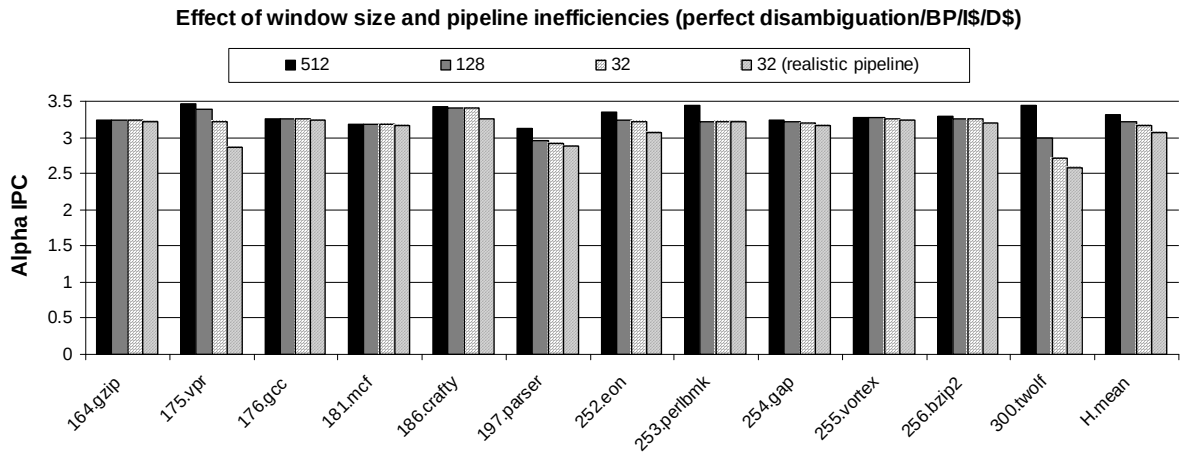



**Figure 7-2 Effect of window size and pipeline ineffeciencies**

Figure 7-2 confirms the validity of the stead-state performance model. The IPC drop of about 0.7 in the most aggressive configuration (the leftmost bars in the chart) from the ideal steady-state IPC of 4 is explained with the following imperfections:

- The instruction fetch stream is limited by the I-cache sub-block size and branch (mis)alignments within the fetched sub-block.
- Multi-cycle functional unit latencies, especially the three cycle load-to-use latency.

Note the relatively small performance difference (less than 3%) between an idealistically wide pipeline configuration with a 32-entry, unified issue queue and the realistically configured pipeline. To some extent, this shows the complexity-effectiveness of the modeled pipeline.

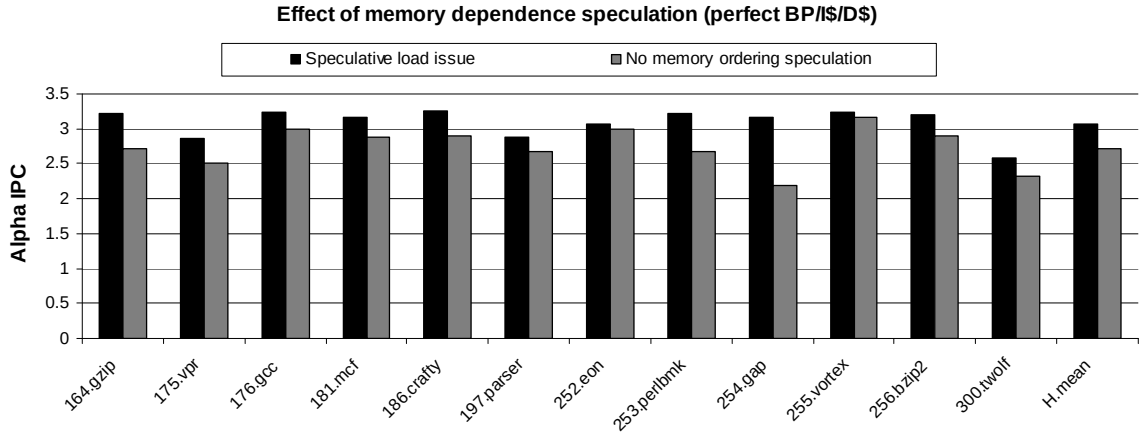



**Figure 7-3 Effect of memory dependence speculation**

It is interesting to see the effect of memory disambiguation schemes in Figure 7-3. By default, the baseline pipeline speculates on the memory dependences (the first bars). In the second configuration, load instructions are not allowed to issue if there is any older store instruction whose address has not be resolved yet. To some extent, this confirms the observation by Moshovos and Sohi [167] that memory dependences can limit the performance of a program substantially. Many current generation high performance processors allow loads to issue speculatively without fully checking all older stores.

Finally, Figure 7-4 shows the effect of Alpha ISA NOPs. Here the average numbers of decoded instructions and the retired instructions per cycle are shown. As explained in the previous chapter, the Alpha ISA NOPs are recognized and removed in the decode stage. On average, 7% of the total IPC is lost due to those NOPs.

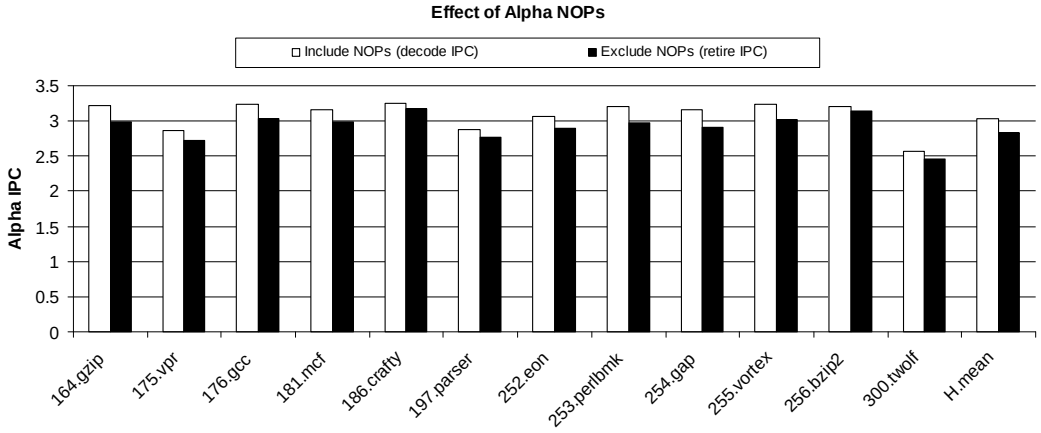



**Figure 7-4 Effect of Alpha NOPs**

This is an example of the binary compatibility requirement restricting the performance of later generation designs. Most of the NOPs were generated by the compiler in an effort to align the branch/jump target addresses, hoping to improve the fetch efficiency. Even though newer designs typically have reasonably deep front-end buffers to squash the fetch-induced bubble cycles, their fetch bandwidths are still artificially reduced by the older program binaries – an unfortunate legacy. The superblock-based dynamic binary translation naturally removes those NOPs.

### 7.2.2 Effect of Mispredictions and Cache Misses

Figure 7-5 shows the average number of costly pipeline events, such as branch mispredictions and cache misses. These events largely determine the overall performance of a machine [127]. Of these events, the L1 cache misses can be partially hidden by reasonably deep pipeline buffers and out-of-order processing as long as the misses are not too frequent. In general, branch mispredictions are relatively more costly than the L1 cache misses. Nonetheless, the L2 cache misses have the biggest effect on the performance if the number of misses is not trivial. From Figure 7-5, it can be inferred that the performance of *181.mcf* will be substantially lower than other benchmarks.

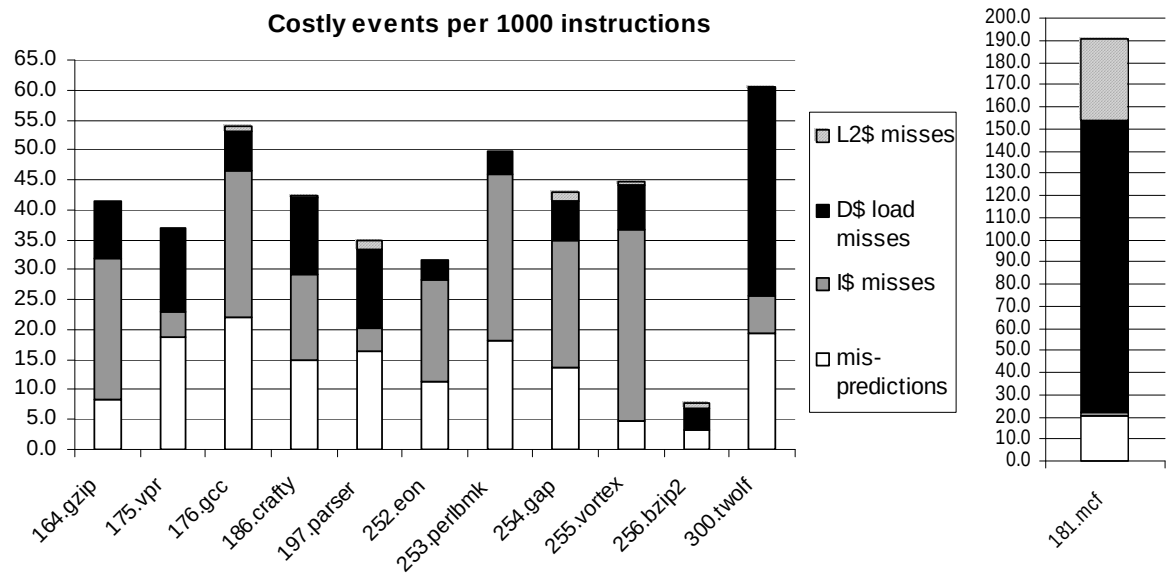



**Figure 7-5 Average number of mispredictions and cache misses per 1,000 instructions**

In Figure 7-6, realistic branch predictors, the I-cache, the D-cache, and the L2-cache are introduced in sequence. It would be interesting to see other combinations such as perfect branch prediction coupled with imperfect caches. Unfortunately, it is very difficult to use such combinations because of the way the simulator was built.

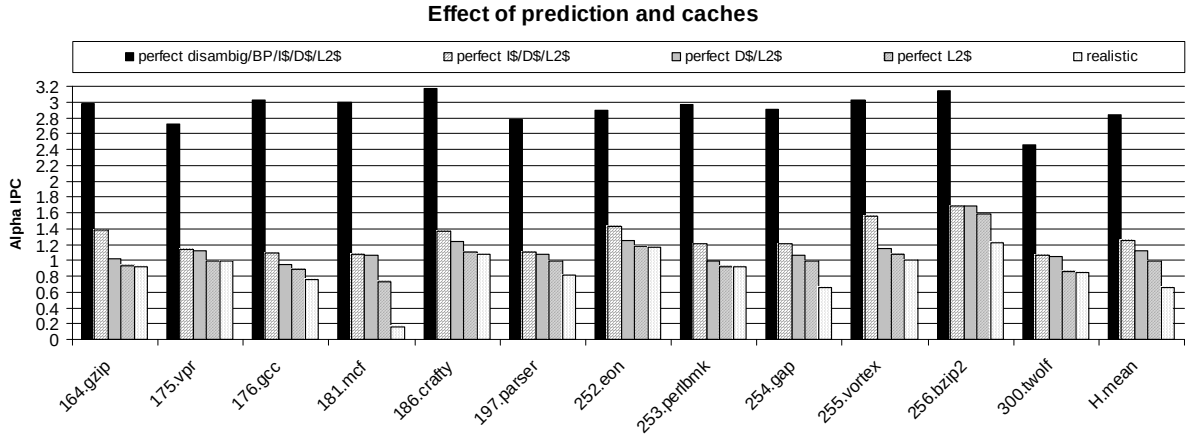



**Figure 7-6 Effect of mispredictions and cache misses**

From Figure 7-6, it is apparent that the branch mispredictions limit the overall performance to the average IPC of about 1.2. Level 1 cache misses do not affect the performance much; this is mostly due to the small working set sizes of the SPEC benchmark programs. The out-of-order instruction execution and various pipeline buffers are able to hide most of those miss events for this workload. However, they are not sufficient to hide L2 cache miss latencies. Those benchmarks that experience a handful of L2 cache misses per 1,000 instructions show a fairly large performance drop. The effect of L2 cache misses is responsible for the low performance of *181.mcf*, to the point where the average IPC performance (in harmonic mean) is largely determined by the performance of that particular benchmark.

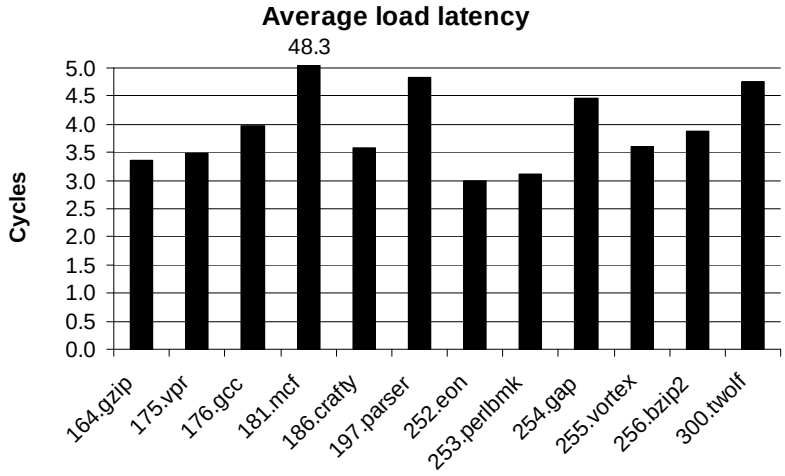



**Figure 7-7 Average load latency**

Figure 7-7 shows the measured average latencies of the load instructions. Again, *181.mcf* has an order of magnitude higher average latency. This average load latency was obtained *after* introducing combining mechanisms in key memory buffers, i.e., LMQ, WB and MSHR. Typically cache misses are "bursty" and show certain amount of spatial locality [127]. If the memory buffers do not perform combining, those characteristics are not exploited and hence, the average load latency is unacceptably high.

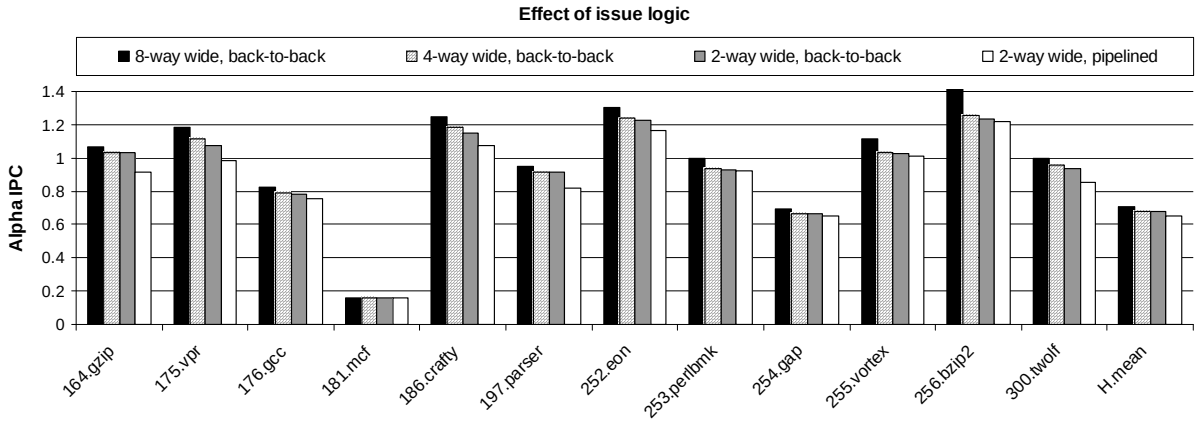



**Figure 7-8 Effect of issue logic**

Lastly, the complexity-effectiveness of the issue logic is shown in Figure 7-8. Here the issue widths for particular functional unit types are varied. For example, in the 2-way wide configuration, maximum two integer, two memory, and two floating-point instructions can be issued as in Figure 6-2. The last configuration uses a pipelined issue logic as in POWER4 – there is one bubble cycle between a single cycle (in terms of the computation latency) instruction and a dependent instruction. Although wider issue does help in some benchmarks (especially *256.bzip2* which is not constrained by branch mispredictions and cache misses), it may not be worth the added complexity of the extra bypass network. On the other hand, the effect of back-to-back issue is not very high (3.7% IPC drop). In other words, the 4-way segmented issue window used in the baseline pipeline is highly complexity-effective.

### 7.2.3 Summary of Baseline Model Evaluation

From the experiments on the baseline pipeline that are modeled after current generation superscalar processors, it is clear that the overall performance is largely determined by the branch prediction accuracy and the memory subsystem design. In other words, given the prediction and

cache performances, it makes better sense to make the “main” pipe relatively simple and reduce the number of mini replays to reduce energy consumption.

## 7.3 Evaluation of the ILDP System

### 7.3.1 Machine Configurations

Table 7-2 summarizes the simulated machine parameters for both the baseline superscalar processor and the ILDP microarchitecture. The default parameters for the ILDP pipeline are shown in bold and used throughout the evaluation unless explicitly stated otherwise. In summary, the default ILDP configuration uses a 4-instruction wide pipeline front-end, 8 logical accumulators, 8-way FIFOs, a two-way replicated 32KB L1 D-cache organized as in Figure 3-6(c), and 0-cycle global communication latency.

**Table 7-2 Simulated machine parameters**

| | Baseline superscalar | ILDP |
|---|---|---|
| L1 I-cache | Direct mapped, 32-KB, 128-byte line size, 1 shared read/write port | |
| L1 D-cache | Fully shared, 32-KB | 1 or **2** instances, **32-KB** or 16-KB |
| | 2-cycle (3-cycle load-to-use), 2 read ports, 1 write port<br>4-way set-assoc., tree-style pseudo LRU, 128-byte line size | |
| L2 cache | 8-cycle for the critical word, 4 bursts to fill L1 caches,<br>runs at half frequency of the main pipeline<br>4-way set-assoc., tree-style pseudo LRU, 1-MB, 128-byte line size, 1 read/write port<br>8-entry combining LMQ, 12-entry combining WB, single-entry FMQ | |
| Memory interface | 96-cycle for the critical L2 cache sub-block, 4 bursts to fill L2 cache line,<br>runs 8 times slower than the main pipeline<br>16-entry combining MSHR | |
| Fetch bandwidth | 32 byte | |
| Instruction align/scan | Max 8 instructions | Max 16 instructions |
| Decode/rename bandwidth | 4 instructions | **4** or 6 instructions |
| Branch prediction | 16K entry, 12-bit global history g-share predictor<br>32-entry RAS<br>512-entry, 4-way set-assoc. BTB<br>256-entry, 4-way set-assoc. JTLT (ILDP only)<br>2 predictions per cycle in the decode stage<br>out-of-order branch resolution, predictors are trained in the retire stage | |
| Store queue | Shared, 32-entry shared, 2 CAM ports | Local, 32-entry, 1 CAM port |
| Other ordering maintenance buffers | 128-entry reorder buffer: 1 branch, 1 store, otherwise 4 instructions per cycle retire bandwidth<br>48-entry load queue,<br>two 80-entry register scoreboards (for integer and fp), | |
| Issue queue | 4 * 8-entry int./mem. out-of-order queue<br>4 * 8-entry fp out-of-order queue | Maximum two instructions are dispatched to a FIFO in the same cycle,<br>4, 6 or **8** 32-entry single issue FIFO queue |
| | Max 2 int, 2 mem, 2 fp instructions | Max 4, 6 or **8** instructions (any type) |
| | Load hit speculation (2-cycle shadow) | No load hit speculation |
| Physical register file | Shared, 80-entry integer, 80-entry fp | Local, 80-entry integer, 80-entry fp |
| Functional units | Fully pipelined, fully shared | Sequential, local within PE |
| | 2 integers, 2 fps, 2 L1 D-cache read ports, 1 write port | 4/6/**8** integers, 4/6/**8** fps, 1/**2** L1 D-cache read ports, 1 broadcast write port |

### 7.3.2 Dynamic Binary Translation Characteristics

**Table 7-3 Translated instruction characteristics**

| Benchmark | Total number of dynamic Alpha instructions | % of Alpha instructions executed in native mode | Relative number of dynamic instructions | % of copy instructions | % of effective address calculation instructions |
|---|---|---|---|---|---|
| *164.gzip* | 3.25 billion | 99.99 | 1.33 | 62.97 | 21.04 |
| *175.vpr* | 1.44 billion | 99.94 | 1.33 | 55.74 | 32.62 |
| *176.gcc* | 1.77 billion | 98.78 | 1.32 | 60.84 | 36.86 |
| *181.mcf* | 4.00 billion | 99.77 | 1.31 | 40.13 | 37.79 |
| *186.crafty* | 4.00 billion | 99.36 | 1.42 | 45.67 | 46.66 |
| *197.parser* | 3.92 billion | 99.99 | 1.34 | 52.66 | 24.75 |
| *252.eon* | 1.72 billion | 99.89 | 1.35 | 39.29 | 46.37 |
| *253.perlbmk* | 4.00 billion | 99.96 | 1.29 | 49.38 | 30.93 |
| *254.gap* | 1.11 billion | 99.98 | 1.26 | 58.01 | 20.16 |
| *255.vortex* | 4.00 billion | 99.40 | 1.25 | 50.08 | 39.51 |
| *256.bzip2* | 4.00 billion | 99.90 | 1.42 | 57.72 | 23.86 |
| *300.twolf* | 4.00 billion | 99.95 | 1.35 | 56.75 | 19.93 |
| **Average** | | **99.74** | **1.33** | **57.88** | **34.84** |

From the 3rd column in Table 7-3, it can be seen that most of the V-ISA instructions are executed in the native execution mode. If the percentage falls below 99% as with *176.gcc*, the impact of the interpretation overhead becomes significant. The effect of the interpretation overhead is evaluated in section 7.3.3.3. The 4th column in Table 7-3 shows the dynamic instruction count expansion phenomenon. On average 33% more instructions are generated. Copy instructions take more than half of those extra instructions. A more elaborate ISA format that supports `GPR op Immediate` mode could reduce the number of extra copy instructions somewhat. In return, the instruction decoding would be much more complex. The last column shows that even with the specialized ISA support for suppressing unnecessary decomposition of a V-ISA instruction into multiple I-ISA instructions, some memory instructions still generate a separate address calculation

instruction. This is due to the reduction of displacement field width from 16-bit (Alpha ISA) to 7-bit (ILDP ISA).

**Table 7-4 Translated instruction characteristics, continued**

| Benchmark | Average instruction size in bytes | Average instruction decode rate | | Total translation cache size | Relative number of static instruction bytes | Number of Alpha instructions to translate a source Alpha instruction |
|---|---|---|---|---|---|---|
| | | Alpha/ baseline | ILDP/ ILDP | | | |
| *164.gzip* | 3.50 | 3.44 | 3.75 | 70,972B | 1.33 | 871.25 |
| *175.vpr* | 3.59 | 3.42 | 3.72 | 101,932B | 1.37 | 878.37 |
| *176.gcc* | 3.57 | 3.25 | 3.62 | 2,466,640B | 1.31 | 870.40 |
| *181.mcf* | 3.68 | 3.17 | 3.50 | 35,152B | 1.50 | 896.92 |
| *186.crafty* | 3.47 | 3.43 | 3.72 | 441,276B | 1.31 | 874.64 |
| *197.parser* | 3.60 | 3.21 | 3.52 | 430,920B | 1.28 | 856.33 |
| *252.eon* | 3.68 | 3.43 | 3.68 | 136,608B | 1.71 | 871.13 |
| *253.perlbmk* | 3.58 | 3.33 | 3.66 | 90,940B | 1.14 | 860.20 |
| *254.gap* | 3.58 | 3.31 | 3.68 | 510,472B | 1.24 | N/A |
| *255.vortex* | 3.59 | 3.36 | 3.72 | 871,796B | 1.23 | 831.44 |
| *256.bzip2* | 3.59 | 3.38 | 3.53 | 33,424B | 1.23 | 864.87 |
| *300.twolf* | 3.62 | 3.41 | 3.73 | 145,376B | 1.14 | 879.20 |
| **Average** | **3.59** | **3.35** | **3.65** | | **1.32** | **858.5** |

On the other hand, the average instruction size (shown in the 2nd column in Table 7-4) is somewhat reduced to 3.59 bytes (from 4 bytes) due to the use of 16-bit instructions. The average instruction decode rates (shown in the 3rd and 4th columns) are obtained by counting the number of decoded instructions in cycles when instruction decoding is not stalled. Due to the combination of smaller average instruction size and the dynamic code re-layout effect, the average number of instructions that are fetched and decoded in the ILDP system is higher than a comparable superscalar pipeline.

The 5th column contains the total code size of all translations in the translation cache. The small working set sizes of the SPEC CPU2000 benchmarks justify the simulation framework design

decision to not implement a translation cache management algorithm. The code expansion rate in the 6th column is the footprint ratio of the all translated code in the translation cache over their corresponding V-ISA superblocks. On average, the code expansion rate is limited to 1.32. The VLIW-based DAISY system reported 1.27 to 7.95 code expansion rates for select SPECint95 benchmarks [70].

On average, less than 860 Alpha instructions were executed to translate a single Alpha instruction (the 7th column in Table 7-4). This number is much less than a quarter of the 4,000+ PowerPC instructions needed to translate a PowerPC instruction to the VLIW architecture, as reported by the DAISY project [70]. Most of this reduction comes from the simple nature of the dynamic binary translation algorithm; no aggressive optimization is performed. All in all, the ILDP dynamic binary translation system achieves fast translation and small translated code size.

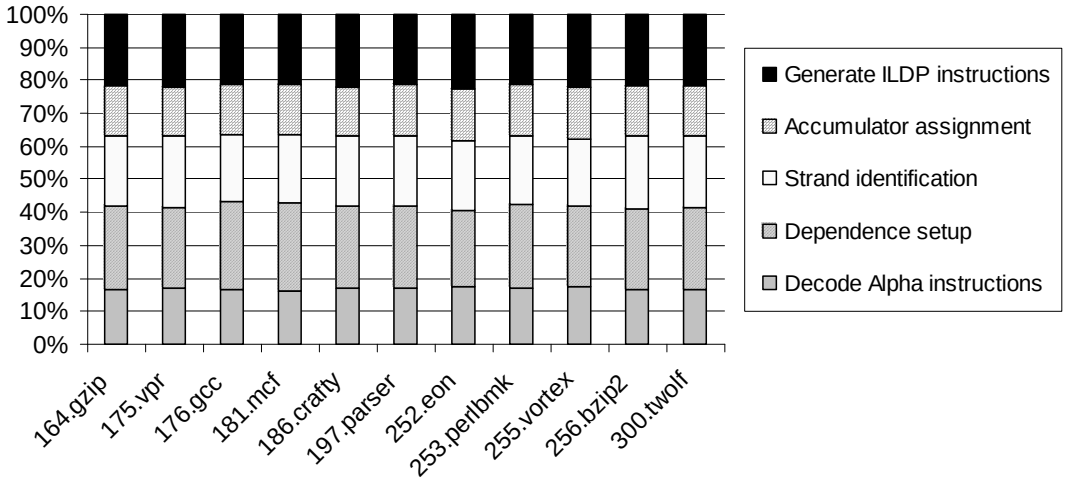



**Figure 7-9 Breakdown of binary translation components**

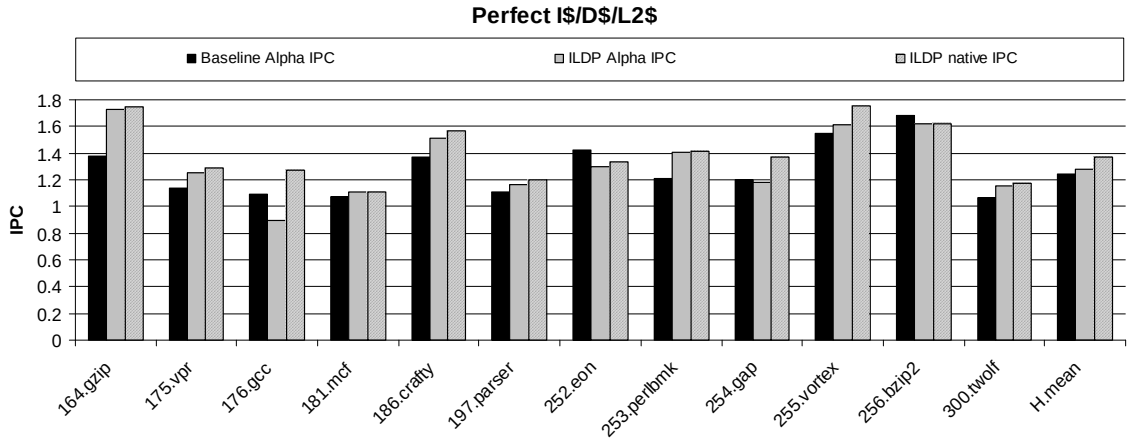
Figure 7-9 shows the breakdown of the binary translation components as number of dynamic instructions executed in each component. Note that Alpha ISA decoding, even with a highly optimized code, takes about 17% of the total dynamic instructions. Similarly the Dynamo project reported that about 25% of the static translator code (total of 265KB) is for decoding the HP

PA-RISC ISA [14]. The significance of the source ISA decoding is directly related to the interpretation overhead.

## 7.3.3 Performance of the ILDP System

### 7.3.3.1 IPC Performance

Figure 7-10 compares performance of the ILDP processor running dynamically translated ILDP I-ISA code with the conventional baseline superscalar processor running original Alpha V-ISA binaries. In all charts, the first two bars, baseline Alpha IPC and ILDP Alpha IPC respectively, represent performance measured with the number of Alpha instructions per cycle. Here *all* V-ISA instructions are accounted; the ILDP Alpha IPC is calculated by dividing the all Alpha instructions (interpreted or natively executed as ILDP instructions) with the total execution time. In other words, the interpretation and translation overheads were included. An average of 870 cycles and 20 cycles were used for translating and interpreting a single Alpha instruction, respectively. The effect of interpretation speed is further discussed in section 7.3.3.3. On the other hand, the ILDP native IPC (represented by the third bars) is calculated as the number of ILDP instructions executed in the native mode divided by the cycles spent in the native mode.



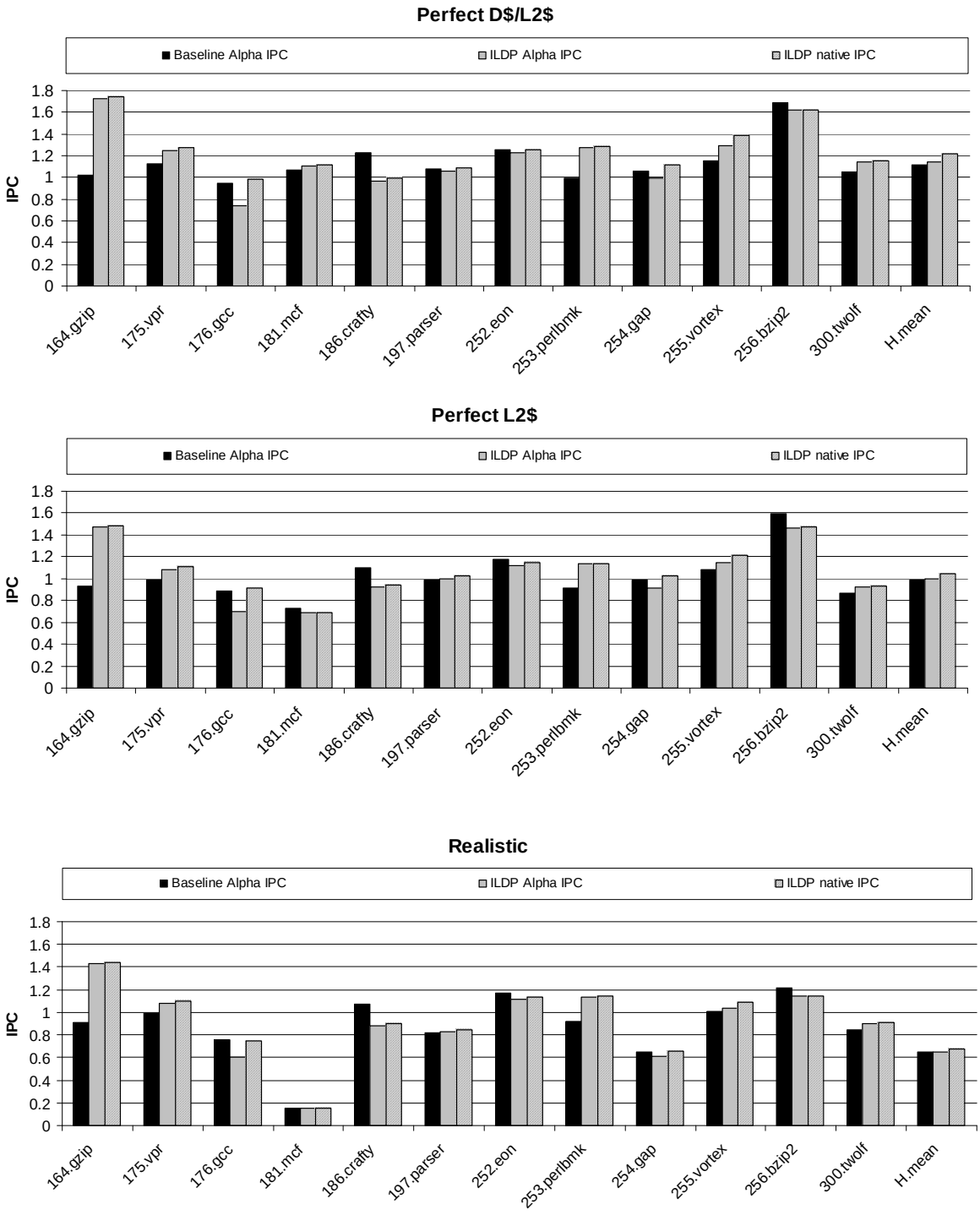



**Figure 7-10 IPC comparisons**

As with the validation method used for the baseline model, the performance evaluation for the ILDP system starts with the idealized conditions. In the first chart in Figure 7-10 where perfect caches are used (but with realistic branch predictors), the ILDP VM system actually *outperforms*, albeit with a small margin of 2.8%, the baseline superscalar processor with a comparable pipeline depth, even without counting the potential clock speed advantage. This IPC performance improvement comes from a combination of the following factors:

- Enhanced fetch efficiency from dynamic code re-layout
- Increased number of functional units
- Reduction of average L1 D-cache access latency from loads without address calculation
- Dramatic reduction of load-related mini replays

The 18.3% IPC drop in *176.gcc* comes mostly from the interpretation overhead, not from microarchitectural inefficiencies of the ILDP system. This can be seen from the 16.5% higher native ILDP IPC compared with the native Alpha IPC of the baseline pipeline.

As realistic caches are introduced, the IPC performances of the two systems start to converge. This is to be expected; in general, a machine's performance is largely determined by costly pipeline flushes (such as branch mispredictions) and cache misses (especially L2 cache misses). An exception to this trend is *186.crafty* whose performance characteristic changes from improvement (with a perfect I-cache) to degradation (with a realistic I-cache). This is explained by the increased number of I-cache misses (shown in Figure 5-8). Of all twelve benchmarks, *176.gcc* and *186.crafty* suffers more I-cache misses. Similarly the performance improvement in *164.gzip* increases from 24.6% (with a perfect I-cache) to 68.6% (with a realistic I-cache) because the code re-layout effect removes I-cache trashing.

Finally, when a realistic L2 cache (and related memory buffers) is used (the last chart), the average IPC performance is largely dominated by *181.mcf* (0.16 IPC) which misses in the L2 cache often. Table 7-5 contains the same IPC numbers in a tabular form.

**Table 7-5 IPC comparisons**

| Bench-mark | Perfect I-cache/D-cache/L2 cache | | | Perfect D-cache/L2 cache | | | Perfect L2 cache | | | Realistic | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Baseline Alpha IPC | ILDP Alpha IPC | ILDP native IPC | Baseline Alpha IPC | ILDP Alpha IPC | ILDP native IPC | Baseline Alpha IPC | ILDP Alpha IPC | ILDP native IPC | Baseline Alpha IPC | ILDP Alpha IPC | ILDP native IPC |
| *164.gzip* | 1.38 | 1.72 | 1.74 | 1.02 | 1.72 | 1.74 | 0.93 | 1.47 | 1.49 | 0.91 | 1.43 | 1.44 |
| *175.vpr* | 1.14 | 1.26 | 1.29 | 1.12 | 1.25 | 1.28 | 0.99 | 1.08 | 1.11 | 0.99 | 1.08 | 1.10 |
| *176.gcc* | 1.09 | 0.89 | 1.27 | 0.95 | 0.75 | 0.99 | 0.88 | 0.70 | 0.91 | 0.76 | 0.57 | 0.99 |
| *181.mcf* | 1.07 | 1.11 | 1.11 | 1.07 | 1.11 | 1.11 | 0.72 | 0.69 | 0.69 | 0.16 | 0.16 | 0.16 |
| *186.crafty* | 1.37 | 1.51 | 1.57 | 1.23 | 0.97 | 0.99 | 1.10 | 0.92 | 0.94 | 1.07 | 0.88 | 0.90 |
| *197.parser* | 1.11 | 1.17 | 1.20 | 1.08 | 1.06 | 1.08 | 0.99 | 1.00 | 1.03 | 0.82 | 0.83 | 0.85 |
| *252.eon* | 1.43 | 1.29 | 1.33 | 1.26 | 1.23 | 1.26 | 1.17 | 1.12 | 1.14 | 1.17 | 1.11 | 1.14 |
| *253.perl-bmk* | 1.21 | 1.40 | 1.42 | 0.99 | 1.27 | 1.28 | 0.92 | 1.13 | 1.14 | 0.92 | 1.13 | 1.14 |
| *254.gap* | 1.20 | 1.18 | 1.37 | 1.06 | 0.99 | 1.12 | 0.99 | 0.92 | 1.02 | 0.65 | 0.61 | 0.66 |
| *255.vortex* | 1.55 | 1.61 | 1.75 | 1.15 | 1.30 | 1.39 | 1.08 | 1.14 | 1.21 | 1.01 | 1.04 | 1.09 |
| *256.bzip2* | 1.68 | 1.62 | 1.62 | 1.68 | 1.62 | 1.62 | 1.59 | 1.46 | 1.47 | 1.22 | 1.14 | 1.14 |
| *300.twolf* | 1.07 | 1.16 | 1.17 | 1.05 | 1.14 | 1.16 | 0.87 | 0.92 | 0.93 | 0.85 | 0.90 | 0.91 |
| **Harmonic mean** | **1.25** | **1.28** | **1.37** | **1.11** | **1.15** | **1.22** | **0.98** | **0.99** | **1.05** | **0.65** | **0.65** | **0.69** |

### 7.3.3.2 Performance Variations over Machine Parameters

In this section, the ILDP microarchitecture machine parameters are changed to estimate their impact on the performance. In Figure 7-11, both front-end and back-end widths are varied. The default configuration (a 4-wide front-end feeding an 8-FIFO-wide back-end) is represented by the second bar (in black). All configurations use four global buses to keep the physical registers coherent.

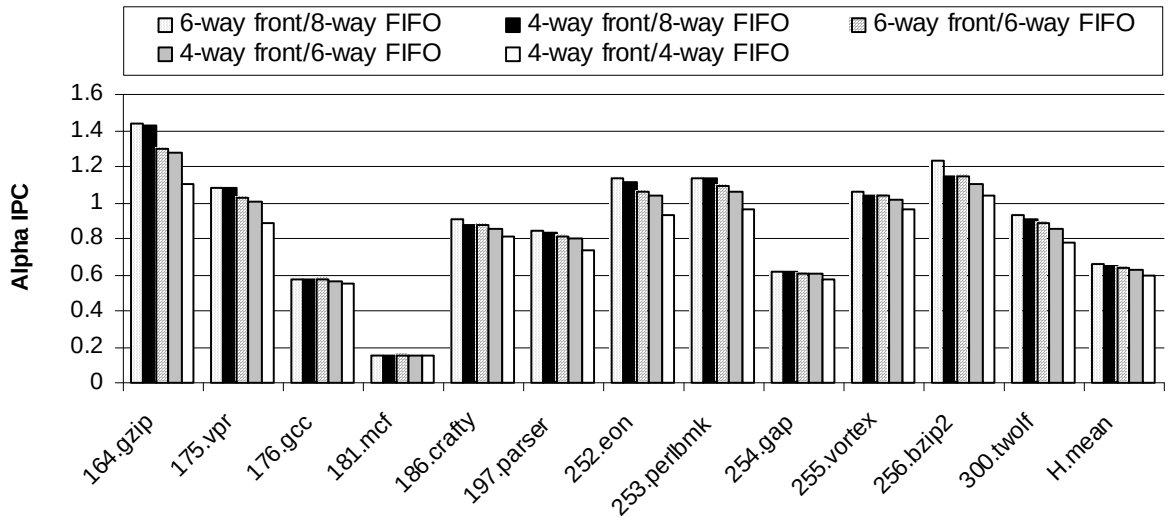



**Figure 7-11 Effect of machine width**

First, the front-end width is increased to 6 instructions (the first bars). This is to make up for the dynamic instruction count expansion. Some benchmarks, especially *256.bzip2*, benefits from this change. However, increasing the front-end width will invariably increase the complexities of the rename logic and especially the steering (accumulator renaming) logic and hence, may not be desirable. On the other hand, the third combination of 6-way front-end and 6-way back-end results in lower overall performance compared to the default configuration. Note that the fourth combination of 4-way front-end and 6-way back-end holds up well (a 2.93% IPC drop compared

with the default configuration). Reducing the number of FIFOs to four leads to a fairly large performance loss in some benchmarks. On average, a 7.77% IPC loss is observed for this configuration. Table 7-6 contains the same IPC performance numbers as the Figure 7-11, but in a tabular form.

**Table 7-6 Effect of machine width**

| Benchmark | Alpha IPC | | | | |
|---|---|---|---|---|---|
| | 8-way FIFO back-end | | 6-way FIFO back-end | | 4-way FIFO |
| | 6-way front-end | 4-way front-end | 6-way front-end | 4-way front-end | 4-way front-end |
| *164.gzip* | 1.44 | 1.43 | 1.30 | 1.27 | 1.10 |
| *175.vpr* | 1.08 | 1.08 | 1.02 | 1.01 | 0.89 |
| *176.gcc* | 0.57 | 0.57 | 0.57 | 0.56 | 0.55 |
| *181.mcf* | 0.16 | 0.16 | 0.16 | 0.16 | 0.15 |
| *186.crafty* | 0.90 | 0.88 | 0.88 | 0.85 | 0.81 |
| *197.parser* | 0.84 | 0.83 | 0.81 | 0.80 | 0.74 |
| *252.eon* | 1.13 | 1.11 | 1.06 | 1.04 | 0.93 |
| *253.perlbmk* | 1.14 | 1.13 | 1.09 | 1.06 | 0.96 |
| *254.gap* | 0.62 | 0.61 | 0.61 | 0.60 | 0.58 |
| *255.vortex* | 1.05 | 1.04 | 1.04 | 1.02 | 0.96 |
| *256.bzip2* | 1.23 | 1.14 | 1.15 | 1.11 | 1.04 |
| *300.twolf* | 0.93 | 0.90 | 0.88 | 0.85 | 0.78 |
| **Harmonic mean** | **0.66** | **0.65** | **0.64** | **0.63** | **0.60** |

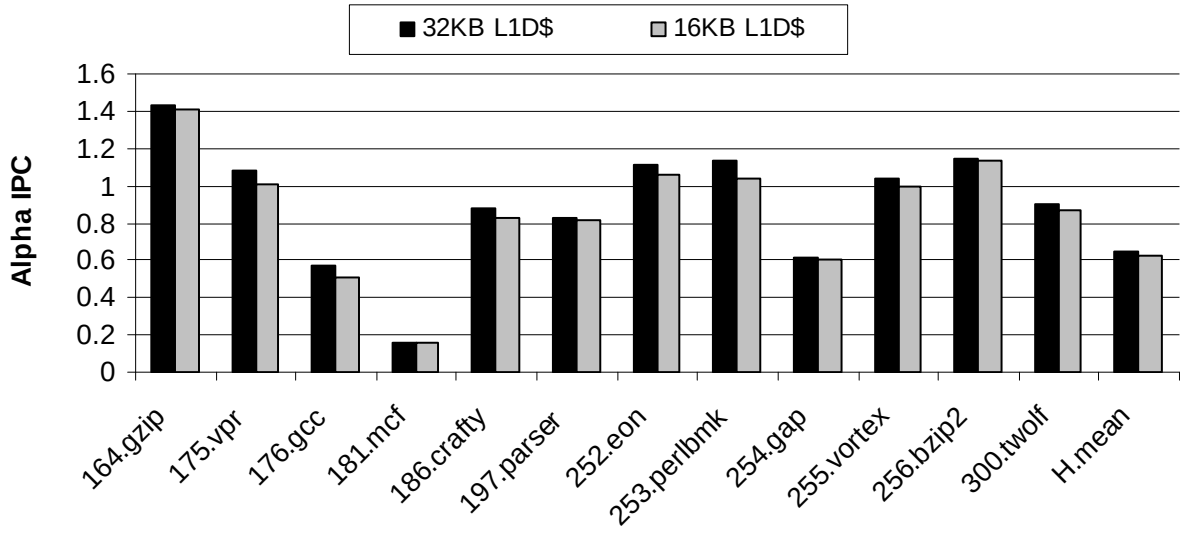



**Figure 7-12 Effect of L1 D-cache size**

The default configuration uses a twice replicated 32KB L1 D-cache to increase the number of read ports to the issue FIFOs (total of four read ports). Although the silicon area of the L1 D-cache is becoming less important, especially with rapidly increasing L2 cache size, cache replication does increase static power consumption due to the transistor count increase. If this is a concern, a smaller L1 D-cache might be considered. In the second configuration in Figure 7-12, a twice replicated 16KB D-cache is used to make up for the replication. On average the impact of L1 D-cache size is fairly small (3.25% IPC drop), at least for the type of workloads used in the experiments.

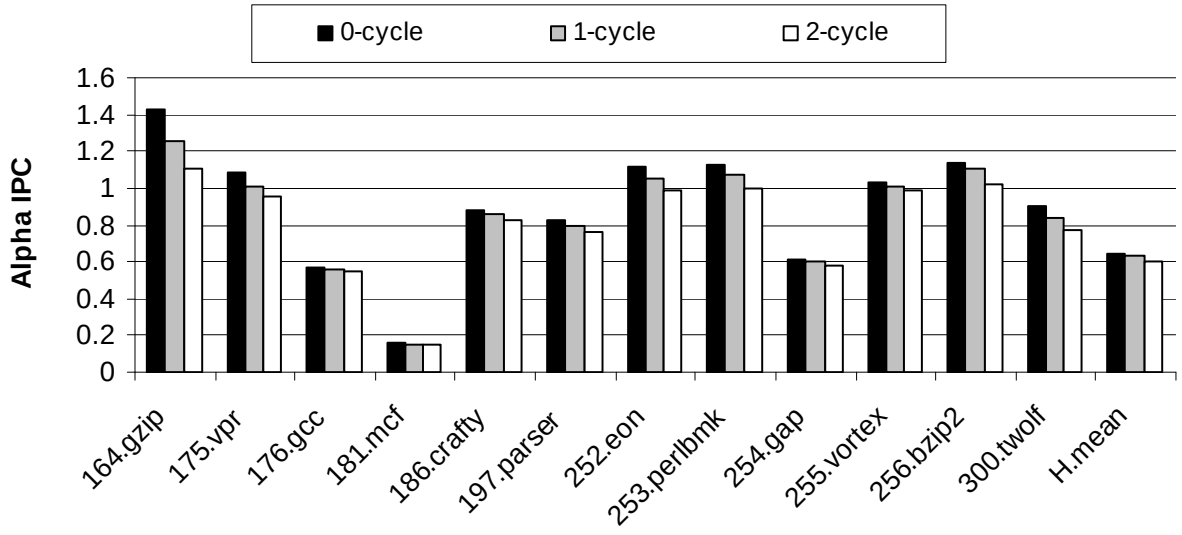



**Figure 7-13 Effect of global wire latencies**

Figure 7-13 shows the performance impact on adding global communication latencies. On average, 2.93% and 6.47% IPC losses are observed for a single cycle and two cycle extra latencies, respectively. This shows that dependence-based strand identification and simple steering based on accumulator numbers works fairly well to tolerate the inter-PE communication latencies.

This performance loss is somewhat larger than what was reported in the early study using the collected program traces [133]. This is mostly due to the increased percentage of global values that are put into the physical registers and kept coherent by the bus network. As was stated in section 4.3.2 and shown in Figure 4-3, the total percentage of instructions that produce global output values in the superblock-based dynamic binary translation is about 40%, a substantial increase compared with about 20% in the program trace study. Here the practical reality of implementation details such as the strict binary compatibility requirement (resulting in increased percentage of global values) and the finite bit-field width mandated by the ISA design (resulting extra copy instructions for `GPR op immediate` mode) are limiting the benefit of the dependence based execution paradigm.

#### 7.3.3.3 Reduction of Mini Replays

In section 3.1.3, it was pointed out that the ILDP microarchitecture provides a unique operand capture model that captures register values from the physical registers (and the physical accumulators) before instructions are issued. As a result, the ILDP issue logic does not speculate on the load instruction latency and hence, mini replays due to latency mis-speculation are completely removed. This leads to dramatic reduction in the number of total mini replays as shown in .

**Table 7-7 Number of total mini replays**

| Benchmark | Number of mini replays per 1,000 instructions | |
|---|---|---|
| | Baseline superscalar | ILDP |
| *164.gzip* | 16.23 | 0.012 |
| *175.vpr* | 26.77 | 0.086 |
| *176.gcc* | 11.89 | 0.071 |
| *181.mcf* | 228.94 | 0.400 |
| *186.crafty* | 15.67 | 0.036 |
| *197.parser* | 55.53 | 0.089 |
| *252.eon* | 9.35 | 0.026 |
| *253.perlbmk* | 7.64 | 0.017 |
| *254.gap* | 14.68 | 0.089 |
| *255.vortex* | 13.75 | 0.023 |
| *256.bzip2* | 8.01 | 0.009 |
| *300.twolf* | 45.61 | 0.162 |
| **Average** | 37.84 | 0.085 |

Note that mini replays are used for other reasons. For example, if an issued load instruction happens to conflict with an incoming L1 D-cache line fill, the load is min-replayed. Note that in the ILDP microarchitecture, only the next instruction in the FIFO needs to be pulled back (if the next instruction starts a new strand and was issued). This will allow a simpler implementation of the mini replay mechanism.

### 7.3.3.4 Impact of Interpretation Overhead

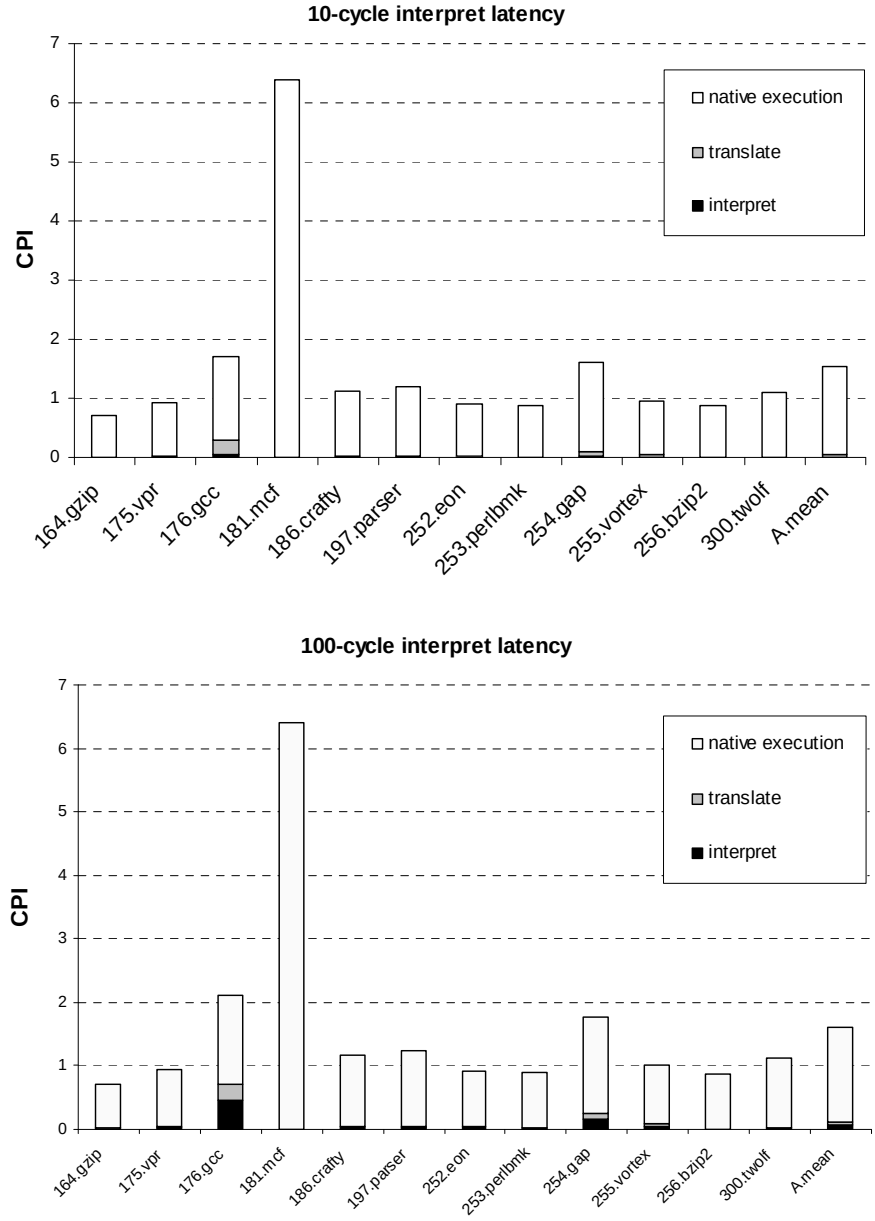



**Figure 7-14 Total CPI breakdown**

In Figure 7-14, average cycles per instruction (CPI) numbers are used instead of IPC. In the first chart, the total CPI is calculated using an aggressive interpretation latency of 10 cycles per source Alpha instruction, while the second chart uses a 100 cycle interpretation latency. From the

chart it can be seen that the translation and interpretation overhead is sufficiently low for most benchmarks. However, some programs do have a large code working set and hence, the virtual machine overhead of interpretation and translation affects the overall performance of those programs significantly. Of the simulated SPEC CPU2000 integer benchmarks, *176.gcc* belongs to those programs. When a 10 cycle interpretation latency is used, that particular benchmark spends 2.7% of the total execution time in the interpretation mode. If a 100 cycle latency (rather large for a simple RISC ISA such as Alpha) is used, *176.gcc* spends 21.6% of the total execution time interpreting newly encountered program codes. In short, the interpreter performance is a very important part of any code cache systems.

### 7.3.4 Complexity Comparisons

Finally, Table 7-8 summarizes the complexity reduction in terms of the number of ports, fan-outs, etc., in hardware structures by showing the baseline superscalar and the ILDP pipelines side-by-side. It can be seen that the ILDP microarchitecture enjoys reduced complexity in all major pipeline hardware structures. For example, the register rename read bandwidth is reduced from 12 to 8 for 4-way wide machines. Issue queue design is dramatically changed; for integer instruction types, a 4-way segmented, 8-entry out-of-order queue that wakes up and selects six instructions in a cycle is replaced with eight (or fewer. depending on the desired performance level) in-order, single-issue FIFO buffers. The number of physical register file read ports were reduced from 7 to 1 by replication. The global result bypass network's fan-out (i.e., capacitive loading) is reduced from 12 to 8 (or less depending on the number of FIFOs) for integer type results.

**Table 7-8 Hardware complexity comparisons**

| | Baseline superscalar | ILDP |
|---|---|---|
| Rename bandwidth | (2 for inputs + 1 for output's old mapping) * 4 = 12 read ports<br>(1 for output's new mapping) * 4 = 4 write ports | (1 + 1) * 4 = 8 read ports<br>1 * 4 = 4 write ports |
| Store queue | Shared, 32-entry shared, 2 CAM ports | Local, 32-entry, 1 CAM port |
| Issue queue | 4 * 8-entry int./mem. out-of-order queue<br>4 * 8-entry fp out-of-order queue | 8 FIFO queue |
| | Max 2 int, 2 mem, 2 fp instructions | Max 4, 6 or 8 instructions (any type) |
| | Load hit speculation (2-cycle shadow) | No load hit speculation |
| Physical register file | Shared | Local |
| | Integer: 2 * 2 (int) + 2 * 1 (mem addr) + 1 (store data) = 7 read ports<br>Fp: 2 * 2 (fp) + 1 (store data) = 5 read ports | Integer: 1 read port<br>Fp: 1 read port |
| | Integer: 2 (int general) + 2 (int load) + 1 (shared) = 5 write ports<br>Fp: 2 (fp general) + 2 (fp load) + 1 (shared) = 5 write ports | Integer: 2 or 4 write ports<br>Fp: 2 or 4 write ports |
| Functional units | Fully pipelined, fully shared | Sequential, local within PE |
| Result bypass network fan-out | Integer: {1(physical register) + 1(ALU)}*[{2(inputs)*2(int general)} + {1(input)*2(load)}] = 12 | Local: 3 (accumulator register)*1 (shared by ALUs and D-cache) + 1 (GPR) = 4 |
| | Int load: (1 + 1)*{(2*2) + (1*2)} = 12 | |
| | Fp load: (1 + 1)*(2*2) = 8 | Global: 8 fan-out,<br>4 results are broadcast per cycle |
| | Fp: {1(physical register) + 1(ALU)} * {2(inputs) * 2(fp issue width)} = 8 | |

### 7.3.5 Summary of the ILDP System Evaluation

From the experiments on the ILDP system that runs a DBT software layer on top of the ILDP microarchitecture, the complexity advantage of the ILDP paradigm was demonstrated. There are two aspects of this. First, the hardware complexity is reduced in most key pipeline structures. In general, reductions in those areas lead to higher clock frequencies, reduced power consumption and silicon area. Reduction of the number of mini replays and simplification in the mini replay mechanism implementation are another areas that show this type of simplifications.

Second, the software complexity (evaluated as the number of instructions required to perform translation) is reduced in the dynamic binary translator. Compared with a previous VLIW-based VM system, the DBT algorithm is more than four times faster. Other types of complexity reduction, such as simpler precise state maintenance, are also achieved.

On average, the overall ILDP system is shown to perform slightly *better*, even including the DBT overheads, than a comparable superscalar design. This is made possible by the combination of many factors, including: improved fetch efficiency from dynamic code re-layout, relatively larger number of functional units, and reduction of the average L1 D-cache hit latency and the number of mini replays. Eventually, however, the overall performance of a system is largely determined by the costly events such as L2 cache misses. Note that this is achieved only after the key virtual machine overheads, such as chaining of register indirect jumps in the code cache, are removed. On the other hand, the dependence-oriented ILDP system is shown to tolerate the global communication latencies fairly well.

# Chapter 8 Conclusions

## 8.1 Thesis Summary

In this thesis, I explored a hardware/software co-designed virtual machine (VM) for instruction-level distributed processing (ILDP). More specifically, I studied and developed three key parts of the co-designed VM system for ILDP, i.e., an accumulator-oriented implementation instruction set format, a distributed, dependence-oriented microarchitecture, and a simple and efficient dynamic binary translation method. Specialized hardware/software support mechanisms for efficient control transfers in a co-designed VM system were also developed. In the thesis, the Alpha instruction set architecture (ISA) is used as the virtual ISA (V-ISA) of the co-designed VM system.

### 8.1.1 Instruction Level Distributed Processing

Instruction-level distributed processing is a design principle that focuses on simple and modular designs with distributed processing at the instruction level. In the ILDP point of view, microarchitecture is considered as a distributed computing problem and communications between instructions are explicitly accounted for as well as computations. Hierarchy is used to contain local communications within a distributed processing element (PE) and manage the overall microarchitecture for global communication between multiple PEs. Previous research [130][171][197][220][224][227] shares the overall distributed processing idea with the ILDP research in this thesis. Combining the following three aspects of the thesis research makes the ILDP paradigm in the thesis unique among the researches on distributed processing. First, the granularity of distributed processing is maintained at the individual instruction level. Second, the

microarchitecture is designed for simplicity, not for pursuing instruction level parallelism (ILP) aggressively. Third, the semantics of the ILDP ISA used in the thesis is sufficiently close to the traditional ISAs, thereby allowing simple and efficient dynamic binary translation to be used for providing binary compatibility with the existing programs.

### 8.1.2 ILDP Instruction Set Architecture

Most existing instruction set architectures are not a good fit for ILDP because typically they do not have a notion of communication hierarchy (other than the most basic distinction between registers and memory). The proposed ILDP ISA in the thesis is based on the following two observations. First, most register values are used only once and most of them are consumed soon after creation; these values are labeled as local. The thesis classifies register value types based on their degree of usage patterns and notes that the dynamic register dependence graph mostly consists of many short chains of dependent instructions, strands, occasionally intersecting with one another. About 70% of all produced values in the collected program traces were found to be local. Second, instructions with true register dependences cannot issue in parallel (unless value speculation techniques are used).

The ILDP ISA has small number of accumulators on top of a relatively large number of general purpose registers (GPRs). The ILDP instruction format assigns temporary values that account for most of the register communications to a small number of accumulators. Only the remaining values (labeled as global) that have long lifetime or multiple consumers are assigned to a larger number of general purpose registers. As a result, the complexity of the register file and associated hardware structures can be greatly reduced.

A strand is a single chain of dependent instructions and identifies the dependence chain’s lifetime. A strand is started with an instruction that has no local input values. Instructions that have

a single local input value assigned to a strand are added to the producer's strand. If an instruction has two local input values, one of the two producer strands is terminated by converting its output value from local to global. There is only one user of a produced value inside a strand. From the program trace study, the average strand size was found to be 2.54 Alpha instructions. An entire strand, sharing a common accumulator, is steered to the ILDP microarchitecture and issued in-order. Strands are issued out-of-order with respect to other strands in different PEs. Accumulator-assigned local values are communicated locally within a PE to the next instruction in the strand. As a consequence, the dependence-oriented ILDP ISA format facilitates simple implementation of a complexity-effective distributed microarchitecture that is tolerant of global communication latencies.

The ILDP ISA format was also designed to be a simple target for a dynamic binary translation (DBT) system. Two major issues in DBT were addressed in the thesis. First, maintaining precise architected state is made relatively easier by the decision that the ILDP DBT system in the thesis does not reschedule instructions; hence the values are produced in the same order as the original program. However, with an accumulator-based ILDP implementation ISA (I-ISA) this is not sufficient because some V-ISA values are held in accumulators and may be overwritten prior to a trap. In the finalized ILDP I-ISA format, every instruction producing a V-ISA result specifies a destination GPR to maintain architected state. This decision affects the second issue of suppressing dynamic code footprint expansion because the destination GPR identifier bit field, coupled with other ILDP-specific bit fields, complicates the ISA format design. Special care was taken to limit the code expansion. For example, a memory instruction format that informs the hardware to dynamically crack the instruction into two micro-instructions was introduced. The ILDP I-ISA includes specialized support instructions for efficient control transfers within a code cache used in the ILDP VM system. The instruction set also uses small number of instruction sizes to offset the instruction count expansion typically found in dynamic binary translation systems.

### 8.1.3 ILDP Microarchitecture

The strand concept of the ILDP ISA is reflected in the accompanying ILDP microarchitecture. The ILDP microarchitecture in the thesis consists of a pipeline front-end of modest width and a number of distributed PEs, each of which performs sequential in-order instruction processing. The ILDP instruction format exposes inter-instruction dependences and local value communication patterns (via accumulator identifiers) to the ILDP microarchitecture, which uses this information to steer instruction strands to the sequential PEs.

The front-end pipeline in the co-designed ILDP microarchitecture is optimized for fetching multiple sequential basic blocks in a cycle. This is to exploit the automatic code re-layout effect of the superblock (a straight-line code sequence with a single entry point and multiple exit points) based dynamic binary translation. GPRs are renamed to physical registers using a traditional rename table. The complexity of the rename logic is reduced because an ILDP instruction only uses a maximum one input GPR and one output GPR. Typical RISC ISA instruction formats use two input GPRs and one output GPR. Accumulators are renamed, or steered, to the issue FIFO buffers. Unlike the traditional register renaming, the old mapping for an accumulator is not required because the accumulator's end of lifetime is explicitly encoded within an instruction. Before the renamed instructions are dispatched to the back-end PEs, instruction ordering enforcement buffers, e.g., reorder buffer, load queue, and store queue, are assigned. The dispatch logic in the ILDP microarchitecture performs the select phase of the traditional out-of-order issue logic and hence, can be made faster. A ring-connected multiple instruction selector logic was developed to fit the dispatch logic's unique requirements – a logic style that scales well with multiple selections is preferred while the circuit speed of the priority encoder is less important due to the omission of the wakeup phase. The physical registers and replicated register scoreboards within PEs perform the wakeup phase of traditional out-of-order issue logic.

Each PE contains an instruction issue FIFO buffer, a local (physical) accumulator, and a local copy of the physical register file. The instructions within a FIFO form a dependence chain, then issue and execute sequentially. Therefore the hardware complexity of a PE is maintained at the level of an in-order scalar design. Because accumulator values stay within the same PE, they can be bypassed without additional delay. The physical register files are kept coherent; their values produced in one PE are communicated to the others through a global communication network. A small number of shared buses are used in the thesis. Taken collectively, a form of out-of-order superscalar execution is achieved by parallel execution of multiple strands.

In the ILDP microarchitecture, instructions read the physical register file before being issued to the functional units. This unique combination of the physical register file based register renaming model and the capture-before-issue operand capture model works to avoid the load latency speculation (typically found in a physical register file based machines) without relying heavily on the bypass networks for operand capturing (as in the machines that rename registers with the reorder buffer or an equivalent mechanism).

Distributing the L1 D-cache is more difficult than the rest of the pipeline. To avoid memory address-based load instruction steering – an extra complexity – from the PEs to the L1 D-cache(s), the ILDP microarchitecture employs a simple combination of sharing and replication for the L1 D-cache. On the other hand, the distributed nature of the ILDP pipeline back-end requires a distributed memory disambiguation solution. The memory instructions are issued speculatively, then check the store and load queues. If an ordering violation is found, the memory queue generates an ordering replay. A simple PC-based dependence predictor is used to limit the number of ordering replays. The latency-critical store queue is replicated for each L1 D-cache read port while the less performance-critical load queue is shared by the multiple PEs. The contents of the replicated store

queue are kept coherent via a separate communication network for memory addresses. A small number of shared busses are used in the thesis.

### 8.1.4 Dynamic Binary Translation for ILDP

For providing binary compatibility with the existing and future programs, the ILDP system in the thesis relies on virtual machine software, co-designed with hardware and hidden from conventional software (including the operating system). The dynamic binary translation subsystem in the virtual machine monitor (VMM) layer maps existing virtual ISA binaries to the ILDP implementation ISA codes on the fly.

One important advantage of the ILDP DBT is the fine-grain state maintenance model. The translated code maintains the same instruction order as the original program; the underlying hardware does reschedule the instructions dynamically but their order is maintained by the collection of ordering enforcement buffers such as the reorder buffer and the store queue. As a result, there is no need to checkpoint the architected state at the translation unit boundaries then roll back and interpret until the trapping instruction is identified, as in previous course-grain co-designed VM systems. This will improve the performance of the ILDP co-designed VM system when a pathologically repetitive trap situation is encountered.

The ILDP DBT system was designed to suppress the code expansion as much as possible. The DBT system uses the 16-bit instruction format if possible, keeps one-to-one instruction mappings as much as possible, exploits known V-ISA idioms, and uses special instructions to suppress instruction count expansion of control transfer instructions.

Unlike previous co-designed VM systems that employed a very long instruction word (VLIW) hardware, the ILDP VM system uses a simplified form of out-of-order superscalar microarchitecture that does not depend heavily on aggressive instruction scheduling by the

translator. As compared with prior VLIW-based co-designed VM systems, the ILDP VM system achieves a new complexity balance point that is closer to hardware.

The main objective of the ILDP DBT algorithm is identifying instruction inter-dependences and making register assignments that reduce inter-PE global communication. First, a dynamic superblock is constructed by collecting the instructions after the superblock start condition is satisfied. The collected instructions are then decoded and put into an intermediate representation (IR) format trace structure. A linear scan of the IR trace is sufficient for setting up the inter-instruction dependences and register usage patterns within the superblock. Next, strand numbers are assigned to the dependence chains based on the input register usage patterns. A finite number of (logical) accumulators are then allocated to the identified strands. The accumulator-assigned IR instructions are converted to the ILDP I-ISA instructions.

Due to the nature of superblock-based translation, all live-in and live-out values are considered as global. Regarding the early exit points (conditional branches) in a superblock, all local values up to an early exit point are converted to global values. This is to avoid complex fix-up codes that recover accumulator-assigned values to GPRs in the newly generated superblocks and the shared dispatch table lookup code. As a result, the percentage of global values in the ILDP DBT rises to about 40% from about 20% in the program trace study.

Over the course of the thesis research, it was found that the traditional superblock chaining method for register indirect jumps was causing substantial performance loss. This loss comes from two places: First, a single jump instruction is replaced with a multiple-instruction compare-and-branch code sequence. If this form of simple software-based prediction is incorrect, control is transferred to a shared dispatch table (a hash table that maps source binary program counter values to translated binary program counter values) lookup code, resulting in even more instruction count expansion. Second, the branch prediction performance is degraded. This is partly due to the sharing

of the dispatch lookup code and partly due to the traditional code cache system's inability to use a hardware return address stack (RAS). Two hardware support mechanisms, jump target-address lookup table (JTLT) and dual-address RAS, were proposed and evaluated. Of the two, the simple dual-address RAS was shown to have the biggest impact on the code cache system's performance.

### 8.1.5 Results and Conclusions

A self-complete simulation infrastructure was developed to evaluate the performance and complexity advantages of the ILDP system. A detailed superscalar microarchitecture simulator that enforces the correctness by design was constructed. This 4-way wide baseline pipeline is largely modeled after IBM POWER4 and models common complexity-effective design trade-offs, various replay mechanisms, and detailed caches and memory subsystems found in modern high performance out-of-order superscalar processors. The ILDP VM system is modeled by integrating the DBT mechanism that changes operating modes between interpretation, translation, and native execution of the translated ILDP codes with a timing simulator that models the ILDP microarchitecture.

As a first step, the baseline pipeline model was validated. This was done by using perfect predictors and caches then making sure the IPC performance is close enough to the expected steady-state IPC performance of 4. During this experiment, it was found that the effect of the instruction issue window organization and other implementation-specific pipeline inefficiencies is much smaller than that of the memory dependence speculation. The ILDP microarchitecture design decisions such as simplified issue mechanisms and use of a memory dependence predictor are in line with these observations. When realistic predictors and caches were introduced, it was found (as expected) that the branch prediction performance and the L2 cache latency largely determine the overall performance. On the other hand, out-of-order execution and deep pipeline buffers were able

to hide L1 cache latencies well. Various issue logic organizations were also evaluated. The POWER4 style 4-way segmented issue logic design was found to be highly complexity-effective, again confirming the insight that the effect of issue logic is of secondary importance compared with other subsystems, especially the L2 cache and the memory interface.

On average, the dynamically translated code shows a 33% increase in the number of instructions and a 32% increase in the total code size. This relatively low code expansion (compared to a VLIW-based co-designed VM) is achieved only after the DBT mechanism is carefully optimized and specialized instruction/hardware support mechanisms such as special control transfer instruction formats, JTLT, and dual-address RAS are employed. On the other hand, the translation overhead was measured by first compiling the whole source code tree of the DBT/timing simulator hybrid framework for the Alpha ISA, then running benchmarks on the framework as usual and measuring the translation overhead as the average number of dynamically executed Alpha ISA instructions for translating a single source Alpha ISA instruction. On average, 858.5 instructions were executed per source instruction. This is less than one quarter of the instructions used in a VLIW-base co-designed VM system.

The IPC performance of the ILDP co-designed VM system is evaluated by dividing all Alpha ISA instructions (either interpreted or natively executed as translated ILDP ISA instructions) with the total execution time (again including both interpretation and translation cycles). A 4-way wide ILDP pipeline with 8 FIFOs sharing a twice-replicated L1 D-cache was used as a default configuration. When perfect caches were used, the ILDP system was shown to perform slightly better (2.8% better IPC) than the baseline superscalar processor of a comparable pipeline depth, without counting the potential clock speed advantage from the complexity reduction in the key pipeline structures. This IPC performance improvement comes from combination of many factors, including enhanced fetch efficiency from the dynamic code re-layout effect of the superblock-based

translation, increased number of functional units making up for the reduced instruction issue opportunities, reduction of the average L1 D-cache access latency from loads without address calculation, and dynamic reduction of load-related mini replays. When realistic caches are used, the average IPC performances of both systems converge. This is mostly due to the effect of the L2 cache misses. This makes good sense; two systems with similar branch prediction and L2 cache behavior should perform similarly, assuming reasonably well-designed pipelines.

Next, the machine parameters were changed to estimate their impact on the ILDP system's performance. As was implied in the instruction count expansion, increasing the front-end width to 6 instructions improves the performance in some benchmarks. However, increasing the front-end width will also increase the complexities of the GPR rename stage and the steering stage (where the accumulators are renamed) and hence, may not be desirable. On the other hand, a combination of a 4-way front-end and a 6-way FIFO back-end holds up fairly well, showing only a 2.94% IPC drop from the default configuration. Reducing the number of FIFOs to 4 leads to a 7.77% IPC degrade.

Adding global result bypass bus latencies of 1-cycle and 2-cycle resulted in 2.93% and 6.47% IPC losses, respectively. This shows that ILDP system is fairly tolerant to the global wire latencies. This performance loss is, however, somewhat larger than expected by the earlier program trace study. This is mostly due to the practical reality of the binary translator implementation details; the simple but strict state maintenance model leads to increase in the percentage of global values in superblocks and the finite bit-field width results in extra copy instructions.

On the other hand, complexity reduction is observed in most key pipeline structures. For example, the register rename read bandwidth is reduced from 12 to 8 for 4-way wide machines. Issue queue design is dramatically changed; for integer instruction types, a 4-way segmented, 8-entry out-of-order queue that wakes up and selects total of six instructions in a cycle is replaced with eight (or less depending on the desired performance level) in-order, single-issue FIFO buffers.

The number of physical register file read ports were reduced from 7 to 1 by replication. The global result bypass network's fan-out level is reduced from 12 to 8 (or less depending on the number of FIFOs) for integer type results.

In summary, the experiments in the thesis show that a co-designed VM system for ILDP performs similarly or better than conventional superscalar processors of similar pipeline depth while achieving lower complexity in key pipeline structures. This complexity reduction in hardware can be exploited to achieve either a higher clock frequency or lower power consumption, or a combination of the two. All in all, the accumulator-oriented ILDP implementation ISA allows a unique combination of a simple, distributed, dynamic microarchitecture and a simple, low-overhead dynamic binary translation, resulting in an efficient implementation of an out-of-order superscalar processor.

## 8.2 Future Research Directions

The hardware/software co-designed virtual machine for instruction level distributed processing has many aspects and this thesis by no means covers them completely. Some of the future research directions are discussed below:

- **Static compilation**: The accumulator-oriented ILDP ISA presents a unique compiler target that requires a different register allocation strategy and quite possibly a different instruction scheduling algorithm. An ILDP compiler, once constructed, will be able to build a data dependence graph and assign accumulators safely. A simple two pass register allocator can also be used as an alternative. In this simpler register allocator, local values within a basic block are assigned to accumulators first. The remaining register values are then assigned to GPRs in the next pass of global register allocation. In any case, the ILDP compiler will favor efficient accumulator assignment over aggressive instruction scheduling. Once

constructed, a compiler for the ILDP ISA will enable further studies on simple and efficient instruction formats. The full potential of the ILDP microarchitecture can also be realized with program binaries explicitly compiled for the microarchitecture. This will be even more interesting with the use of a simpler ILDP ISA format [133][134] that does not contain a destination register identifier (which was introduced in the thesis to support a precise trap model in the virtual machine system). More instructions will be 16-bit and hence, the compiled program binary size is expected to be smaller than a comparable RISC ISA program binary.

- **Reducing interpretation overheads**: Over the course of evaluation, it became apparent that the overhead of the virtual ISA instruction interpretation could be a key performance bottleneck in a co-designed VM system. A fast interpretation mechanism is essential because there are many programs in real-world situations that have a large code working set size or are not repetitive by nature. Efficient helper mechanisms, most likely in the form of relatively simple hardware support structures and associated instructions, will help reduce the virtual ISA decoding overhead.
- **Distributed data cache and memory disambiguation mechanism**: The L1 D-cache is the least developed part of the ILDP microarchitecture in terms of the hardware resource distribution. This is mainly due to the fact that the entire ILDP paradigm in the thesis is built around register dependences, not memory dependences. Nonetheless, extending the ILDP idea to a fully distributed cache subsystem without breaking the overall simplicity of the system is certainly desirable.
- **Floating-point and multi-media type workloads**: In a sense, the ILDP microarchitecture in the thesis makes up the reduced scheduling freedom with more functional units. These multiple functional units are a perfect fit for high ILP workloads typically found in floating-

point and multi-media type programs. Extending the ILDP microarchitecture with a vector-like extension, or even a bit-slice style extension, may be an interesting direction.

- **Further ISA/DBT optimization opportunities**: Although much care was taken to reach good ISA design trade-offs in the thesis, there still can be room for further improvement, especially regarding dynamic instruction count and code size expansion. Some requirements of the strand execution model could be relaxed to allow better dynamic binary translation, for example.
- **Static binary translation**: In general, a 100% compatible static binary translation is not possible. However, for programs that follow a well-defined compiler convention under a well controlled environment (e.g., some embedded systems), static binary translation may be useful, especially for fixed-width V-ISAs. Another possibility is a pseudo-static procedure-level binary translation system, a la FX!32 [112].
- **Other virtual ISAs**: More complex ISAs, especially Intel x86, may be a better target for a co-designed virtual machine paradigm [147] because a complex ISA tends to allow more implementation freedom to the dynamic binary translator, let alone their huge popularity.

# Bibliography


[1] Jaume Abella, Ramon Canal, Antonio Gonzalez, "Power- and Complexity-Aware Issue Queue Designs," *IEEE Micro*, Vol. 23, No. 5, pp. 50-58, Sep. 2003.

[2] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, Brian Gaeke, "LLVA: A Low-Level Virtual Instruction Set Architecture," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 205-216, Dec. 2003.

[3] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, Doug Burger, "Clock Rate vs. IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 248-259, Jun. 2000

[4] Vikas Agarwal, Stephen W. Keckler, Doug Burger, "The Effect of Technology Scaling on Microarchitectural Structures," *Tech Report TR2000-02*, The University of Texas at Austin, 2000.

[5] Pritpal Ahuja, Douglas W. Clark, Anne Rogers, "The Performance Impact of Incomplete Bypassing in Processor Pipelines," *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 36-45, Nov. 1995.

[6] Haitham Akkary, Michael A. Driscoll, "A Dynamic Multithreading Processor," *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 226-236, Nov. 1998.

[7] Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 423-434, Dec. 2003.

[8] Alex Aleta, Josep M. Codina, Antonio Gonzalez, David Kaeli, "Instruction Replication for Clustered Microarchitectures," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 326-335, Dec. 2003.

[9] Erik R. Altman, Michael Gschwind, Sumedh Sathaye, S. Kosonocky, Arthur Bright, Jason Fritts, Paul Ledak, David Appenzeller, Craig Agricola, Zachary Filan, "BOA: The Architecture of a Binary Translation Processor," *IBM Research Report RC 21665*, Dec. 2000

[10] Bharadwaj S. Amrutur, Mark A. Horowitz, "Speed and Power Scaling of SRAM's," *IEEE Transactions on Solid-State Circuits*, Vol. 35, No. 2, pp. 175-185, Feb. 2000.

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

[12] James E. Bahr, Sheldon B. Levenstein, Lynn A. McMahon, Timothy J. Mullins, Andrew H. Wottreng, "Architecture, Design, and Performance of Application System/400 (AS/400) Multiprocessors, *IBM Journal of Research and Development*, Vol. 36, No. 6, pp. 1001-1014, Nov. 1992.

[13] H. B. Bakoglu, Gregory F. Grohoski, Robert K. Montoye, "The IBM RISC System/6000 Processor: Hardware Overview," *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 12-23, Jan. 1990.

[14] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo," *Hewlett Packard Laboratories Technical Report HPL-1999-78*, Jun. 1999.

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

[16] Rajeev Balasubramonian, Sandhya Dwarkadas, David H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 237-248, Dec. 2001.

[17] Rajeev Balasubramonian, Sandhya Dwarkadas, David H. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 275-286, Jun. 2003.

[18] Amirali Baniasadi, Andreas I. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Procesors," *Proceedings of the 33rd International Symposium on Microarchitecture*, pp.337-347, Dec. 2000.

[19] John Banning, H. Peter Anvin, Benjamin Gribstad, David Keppel, Alex Kleiber, Paul Serris, "Fine Grain Translation Discrimination," *US Patent 6,363,336*, Mar. 2002.

[20] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, Yigal Zemach, "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.191-201, Dec. 2003.

[21] Ravi Bhargava, Lizy K. John, "Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 264-274, Jun. 2003.

[22] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, Ben Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 282-293, Jun. 2000.

[23] Robert Bedichek, "Some Efficient Architecture Simulation Techniques," *Proceedings of the USENIX Winter 1990 Technical Conference*, pp. 53-64, Jan. 1990.

[24] Robert Bedichek, "Talisman: Fast and Accurate Multicomputer Simulation," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 14-24, May 1995.

[25] James R. Bell, "Threaded Code," *Communications of ACM*, Vol. 16, No. 6, pp. 370-372, Jun. 1973.

[26] Marc Berndl, Laurie Hendren, "Dynamic Profiling and Trace Cache Generation," *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pp. 276-285, Mar. 2003.

[27] Bryan Black, John P. Shen, "Scalable Register Renaming via the Quack Register File," *Tech Report CmuART-2000-01*, Carnegie Mellon University, 2000.

[28] Darrel Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, K. S. Venkatraman, "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, Vol. 8, Issue 1, pp. 1-17, Feb. 2004.

[29] Eric Borch, Eric Tune, Srilatha Manne, Joel Emer, "Loose Loops Sink Chips," *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pp. 299-310, Feb. 2002.

[30] Shekhar Borkar, "Design Challenges of Technology Scaling," *IEEE Micro*, Vol. 19, No. 4, pp. 23-29. Jul. 1999.

[31] Edward Brekelbaum, Jeff Rupley II, Chris Wilkerson, Bryan Black, "Hierarchical Scheduling Windows," *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 27-36, Nov. 2002.

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

[33] Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows," *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[34] 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.

[35] Douglas C. Burger and Todd M. Austin, "The SimpleScalar Toolset, Version 2.0," *Technical Report CS-TR-97-1342*, University of Wisconsin—Madison, Jun. 1997.

[36] Jeffery A. Butts, Gurindar S. Sohi, "Dynamic Dead Instruction Detection and Elimination," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 199-210, Oct. 2002.

[37] Jeffery A. Butts, Gurindar S. Sohi, "Characterizing and Predicting Value Degree of Use," *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 15-26, Nov. 2002.

[38] Jeffery A. Butts, Gurindar S. Sohi, "Use-Based Register Caching with Decoupled Indexing," *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 302-313, Jun. 2004.

[39] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, Mikko H. Lipasti, "Precise and Accurate Processor Simulation," *Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2002.

[40] Brad Calder, Dirk Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," *Proceedings of the 21st ACM Symposium on Principles and Practices of Programming Languages*, pp. 397-408, Jan. 1994.

[41] Brad Calder, Dirk Grunwald, "Fast & Accurate Instruction Fetch and Branch Prediction," *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 2-11, Jun. 1994.

[42] Ramon Canal, Joan-Manuel Parcerisa, Antonio Gonzalez, "A Cost-Effective Clustered Architecture," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 160-168, Oct. 1999.

[43] Ramon Canal, Joan-Manuel Parcerisa, Antonio Gonzalez, "Dynamic Cluster Assignment Mechanisms," *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp.132-142, Jan. 2000.

[44] Ramon Canal, Antonio Gonzalez, "A Low-Complexity Issue Logic," *Proceedings of the 14th International Conference on Supercomputing*, pp. 327-335, May 2000.

[45] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, David M. Gillies, "Mojo: A Dynamic Optimization System," *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2000.

[46] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, John Yates, "FX!32 – A Profile-Directed Binary Translator," *IEEE Micro*, Vol. 18, No. 2, pp. 56-64, Mar. 1998.

[47] Yuan Chou, John P. Shen, "Instruction Path Coprocessors," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 270-281, Jun. 2000.

[48] 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.

[49] 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.

[50] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, Burzin A. Patell, "Optimization of Instruction Fetch Mechanism for High Issue Rate," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-344, Jun. 1995.

[51] Thomas M. Conte, Sumedh W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 208-218, Dec. 1995.

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

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

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

[55] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, Nigel P. Topham, "Multiple-Banked Register File Architectures," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 316-325, Jun. 2000.

[56] Vinodh Cuppu, Bruce Jacob, "Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?," *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 62-71, Jun. 2001.

[57] Jeffery 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 30th International Symposium on Microarchitecture*, pp. 292-302, Dec. 1997.

[58] Dean Deaver, Rick Gorton, Norman Rubin, "Wiggins/Redstone: An Online Program Specializer," *Proceedings of the 11th HotChips Symposium*, Aug. 1999.

[59] Eddy H. Debaere, Jan M. Van Campenhout, "Interpretation and Instruction Path Coprocessing," *The MIT Press*, 1990.

[60] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, Jim Mattson, "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 Optimization*, pp. 15-24, Mar. 2003.

[61] Rajagopalan Deskian, Douglas C. Burger, Stephen W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 266-277, Jun 2001.

[62] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, Joseph A. Fisher, "DELI: A New Run-Time Control Point," *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 257-268, Nov. 2002.

[63] Keith Diefendorff, "K7 Challenges Intel," *Microprocessor Report*, Vol. 12, No. 14, pp. 1-7, Oct. 1998.

[64] Keith Diefendorff, "Athlon Outruns Pentium III," *Microprocessor Report*, Vol. 13, No. 11, pp. 1, 6-11, Aug. 1999.

[65] Keith Diefendorff, "Hal Makes Sparcs Fly," *Microprocessor Report*, Vol. 13, No. 15, Nov. 1999.

[66] 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.

[67] Evelyn Duesterwald, Vasanth Bala, "Software Profiling for Hot Path Prediction: Less is More," *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 202-211, Nov. 2000.

[68] Kemal Ebcioglu, Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *IBM Research Report RC 20538*, Aug. 1996.

[69] Kemal Ebcioglu, Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 26-37, Jun. 1997.

[70] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, Sumedh Sathaye, "Dynamic Binary Translation and Optimization," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 529-548, Jun. 2001.

[71] Kemal Ebcioglu, Erik R. Altman, Sumedh Sathaye, Michael Gschwind, "Optimizations and Oracle Parallelism with Dynamic Translation," *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 284-295, Nov. 1999.

[72] John H. Edmondson, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, Gilbert M. Wolrich, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal*, Vol. 7, No. 1, pp. 119-135, Jan. 1995.

[73] Dan Ernst, Todd Austin, "Efficient Dynamic Scheduling through Tag Elimination," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 37-46, Jun. 2002.

[74] Dan Ernst, A. Hamel, Todd Austin, "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 253-262, Jun. 2003.

[75] Rolf Ernst, "Codesign of Embedded Systems: Status and Trends," *IEEE Design & Test of Computers*, Vol. 15, No. 2, pp.45-54, Apr. 1998.

[76] M. Anton Ertl, David Gregg, "The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures," *Proceedings of the European Conference on Parallel Computing*, pp. 403-412, Aug. 2001.

[77] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. Patel, 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.

[78] Keith I. Farkas, Norman P. Jouppi, Paul Chow, "Register File Design Considerations in Dynamically Scheduled Processors," *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pp. 40-51, Feb. 1996.

[79] Keith I. Farkas, Paul Chow, Norman P. Jouppi, Zvonko Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 40-51, Dec. 1997.

[80] James A. Farrel, Timothy C. Fischer, "Issue Logic for a 600-MHz Out-of-Order Execution Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol. 33, No. 5, May 1998.

[81] John G. Favor, "RISC86 Instruction Set," *United States Patent 6,336,178*, Jan. 2002.

[82] Brian Fields, Shai Rubin, Rastislav Bodik, "Focusing Processor Policies via Critical-Path Prediction," *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 74-85, Jun. 2001.

[83] Brian Fields, Rastislav Bodik, Mark D. Hill, "Slack: Maximizing Performance under Technological Constraints," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 47-58, May 2002.

[84] Michael J. Flynn, Patrick Hung, Kevin W. Rudd, "Deep Submicron Microprocessor Design Issues," *IEEE Micro*, Vol. 19, No. 4, pp. 11-22, Jul. 1999.

[85] Manoj Franklin, Mark Smotherman, "A Fill Unit Approach to Multiple Instruction Issue," *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 162-171, Dec. 1994.

[86] Manoj Franklin, Gurindar S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 58-67, Dec. 1992.

[87] Manoj Franklin, Gurindar S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Transactions on Computers*, Vol. 45, No. 5, pp. 552-571, May 1996.

[88] Daniel H. Friendly, Sanjay J. Patel, Yale N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 173-181, Dec. 1998.

[89] Peter N. Glaskowsky, "MemoryLogix Makes Tiny x86," *Microprocessor Report*, pp. 1-3, Nov. 11, 2002.

[90] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, Robert C. Valentine, "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology Journal*, Vol. 7, Issue 2, pp. 21-36, May 2003.

[91] Masahiro Goshima, Kengo Nishino, Yasuhiko Nakashima, Shin-ichiro Mori, Toshiaki Kitamura, Shinji Tomita, "A High Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 225-236, Dec. 2001.

[92] R. Govindarajan, Hongbo Yang, Jose Nelson Amaral, Chihong Jhang, Guang R. Gao, "Minimum Instruction Register Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures," *IEEE Transactions on Computers*, Vol. 52, No. 1, pp. 4-20, Jan. 2003.

[93] Michael Gschwind, "Method and Apparatus for Determining Branch Addresses in Programs Generated by Binary Translation, *IBM Disclosures YOR819980334*, Jul. 1998.

[94] Michael Gschwind, "Method and Apparatus for Rapid Re-turn Address Computation in Binary Translation," *IBM Disclosures YOR819980410*, Sep. 1998.

[95] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, David Appenzeller, "Dynamic and Transparent Binary Translation," *IEEE Computer*, Vol. 33, No. 2, pp. 54-59, Mar. 2000.

[96] Michael Gschwind, Erik Altman, "On Achieving Precise Exception Semantics in Dynamic Optimization," *IBM Research Report RC 21900*, Dec. 2000.

[97] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, Jonathan C. Hall, "Managing the Impact of Increasing Microprocessor Power Consumption," *Intel Technology Journal Q1*, 2001.

[98] Linley Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, pp. 9-15, Feb. 16, 1995.

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

[100] A. Hartstein, Thomas R. Puzak, "The Optimal Pipeline Depth for a Microprocessor," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 7-13, Jun. 2002.

[101] A. Hartstein, Thomas R. Puzak, "Optimum Power/Performance Pipeline Depth," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 117-125, Dec. 2003.

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

[103] Kim Hazelwood, Michael D. Smith, "Generational Cache Management of Code Traces in Dynamic Optimization Systems," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 169-179, Dec. 2003.

[104] Timothy H. Heil, James E. Smith, "Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines," *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 281-290, Dec. 2000.

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

[106] Dana Henry, Bradley C. Kuszmaul, Gabriel H. Loh, Rahul Sami, "Circuits for Wide-Window Superscalar Processors," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 236-247, Jun. 2000.

[107] Hewlett Packard Development Company, "PA-RISC 8x00 Family of Microprocessors with Focus on PA-8700," http://www.cpus.hp.com/technical_references/PA-8700wp.pdf

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

[109] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Carmean, Alan Kyker, Patrice Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal Q1*, pp. 1-12, 2001.

[110] 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.

[111] Paul Hohensee, Mathew Myszewski, David Reese, "Wabi CPU Emulation," *Proceedings of the 8th HotChips Symposium*, pp. 47-65. Aug. 1996.

[112] Raymond J. Hookway, Mark A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, Vol. 9, No. 1, pp. 3-12, Jan. 1997.

[113] M. S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Douglas C. Burger, Stephen W. Keckler, Premkishore Shivakumar, "The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 14-24, Jun. 2002.

[114] Jhigang Hu, David Brooks, Pradip Bose, "Microarchitecture-Level Power-Performance Simulators: Modeling, Validation, and Impact on Design," *Tutorial in conjunction with the 36th International Symposium on Microarchitecture*, Dec. 2003.

[115] Jie S. Hu, Narayanan Vijaykrishnan, Mary J. Irwin, "Exploring Wakeup-free Instruction Scheduling," *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pp. 232-243, Feb. 2004.

[116] Shiliang Hu, James E. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions," *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pp. 213-226, Mar. 2004.

[117] Wen-mei W. Hwu, , Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, Kluwer Academic Publishing, pp. 229-248, 1993.

[118] Intel Corp., Intel Itanium Architecture Software Developer's Manual vol. 3, Rev. 2.0: Instruction Set Reference, *Intel Corp.*, 2001.

[119] International Business Machine Corp., "DAISY: Dynamically Architected Instruction Set from Yorktown", http://www.research.ibm.com/daisy/

[120] International Technology Roadmap for Semiconductors, "International Technology Roadmap for Semiconductors 2003 Edition Executive Summary," ITRS, 2003.

[121] Bruce Jacob, Trevor Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, Vol. 18, No. 4, pp. 60-75, Jul. 1998.

[122] Quinn Jacobson, James E. Smith, "Instruction Pre-Processing in Trace Processors," *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 125-129, Jan. 1999.

[123] Daniel A. Jimenez, Stephen W. Keckler, Calvin Lin, "The Impact of Delay on the Design of Branch Predictors," *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 67-76, 2000.

[124] Mike Johnson, Superscalar Microprocessor Design, *Prentice-Hall Inc*., Englewood Cliffs, New Jersey, 1991.

[125] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, Adi Yoaz, "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification," *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 216-225, 1998.

[126] David R. Kaeli, Philip G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 34-42, Jun. 1991.

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

[128] Edmund J. Kelly, Robert F. Cmelik, Malcom J. Wing, "Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed," *US Patent 5,832,205*, Nov. 1998.

[129] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, Pat Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro*, Vol. 23, No. 2, pp. 66-76, Mar. 2003.

[130] Gregg A. Kemp, Manoj Franklin, "PEWS: A Decentralized Dynamic Scheduler for ILP Processing," *Proceedings of the International Conference on Parallel Processing*, pp. 239-246, Aug. 1996.

[131] Richard E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, Vol. 19, No. 2, pp. 24-36, Mar. 1999.

[132] Changkyu Kim, Douglas C. Burger, Stephen W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 211-222, Oct. 2002.

[133] Ho-Seop Kim, James E. Smith, "An Instruction Set and Microarchitecture for Instruction-Level Distributed Processing," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 71-81, Jun. 2002.

[134] Ho-Seop Kim, 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.

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

[136] Ilhyun Kim, Mikko L. Lipasti, "Half-price Architecture," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 28-38, Jun. 2003.

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

[138] Ilhyun Kim, Mikko L. Lipasti, "Understanding Scheduling Replay Schemes," *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pp. 198-209, Feb. 2004.

[139] Thomas Kistler, Michael Franz, "Continuous Program Optimization: Design and Evaluation," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 549-565, Jun. 2001.

[140] Alexander Klaiber, "The Technology behind Crusoe Processors," *Transmeta Technical Brief*, 2000.

[141] Paul Klint, "Interpretation Techniques," *Software Practice and Experience*, Vol. 11, No. 9, pp. 963-973, Sep. 1981.

[142] Steven R. Kunkel, James E. Smith, "Optimal Pipelining in Supercomputers," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 404-411, Jun. 1986.

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

[144] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, Eric Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 59-70, Jun. 2002.

[145] 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*, pp. 27-38, Jun. 1996.

[146] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, Saman P. Amarasinghe, "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine," *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 46-57, Oct. 1998.

[147] Jochen Liedtke, Nayeem Islam, Trent Jaeger, Vsevolod Panteleenko, Yoonho Park, "An Unconventional Proposal: Using the x86 Architecture as the Ubiquitous Virtual Standard Architecture," *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pp. 237-241, Sep. 1998.

[148] Mikko H. Lipasti, John P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226-237, Dec. 1996.

[149] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, John C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing*, Kluwer Academic Publishing, pp. 51-142, 1993.

[150] Luis A. Lozano, C., Guang R. Gao, "Exploiting Short-Lived Variables in Superscalar Processors," *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 292-302, Dec. 1995.

[151] Peter S. Magnusson, David Samuelsson, "A Compact Intermediate Format for SIMICS," *Technical Report R94:17*, Swedish Institute of Computer Science, 1994.

[152] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal Q1*, pp. 1-12, 2002.

[153] Milo M. Martin, Amir Roth, Charles N. Fischer, "Exploiting Dead Value Information," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 125-135, Dec. 1997.

[154] Doug Matzke, "Will Physical Scalibility Sabotage Performance Gains," *IEEE Micro*, Vol. 30, No. 9, pp. 37-39, Sep. 1997.

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

[156] Stephen Melvin, Yale N. Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA," *International Journal of Parallel Programming*, Vol. 23, No.3, pp. 221-243, 1995.

[157] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, Wen-mei W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 136-147, May 1999.

[158] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, Wen-mei W. Hwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 59-70, Jun. 2000.

[159] 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 Run-Time Optimization," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 567-589, Jun. 2001.

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

[161] Teresa Monreal, Antonio Gonzalez, Mateo Valero, Jose Gonzalez, Victor Vinals, "Delaying Physical Register Allocation through Virtual-Physical Registers," *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 186-192, Nov. 1999.

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

[163] Gordon E. Moore, "No Exponential Is Forever: But 'Forever' Can Be Delayed," *Invited Talk, IEEE International Solid State Circuit Conference*, Jan. 2003.

[164] Enric Morancho, Jose Llaberia, Angel Olive, "Recovery Mechanism for Latency Misprediction," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 118-130, Sep. 2001.

[165] Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, Gurindar S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181-193, Jun. 1997.

[166] Andreas Moshovos, Gurindar S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 235-245, Dec. 1997.

[167] Andreas Moshovos, Gurindar S. Sohi, "Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors," *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp. 301-312, Jan. 2000.

[168] Mayan Moudgill, John-David Wellman, Jaime H. Moreno, "Environment for PowerPC Microarchitecture Evaluation," *IEEE Micro*, Vol. 19, No. 2, pp. 15-25, Mar. 1999.

[169] Steven S. Muchnick, "Advanced Compiler Design and Implementation," *Morgan Kaufmann Publishers*, Inc., pp. 726-733, 1997.

[170] Trevor Mudge, "Power: A First-Class Architectural Design Constraints," *IEEE Computer*, Vol. 34, No. 4, pp. 52-58, Apr. 2001.

[171] Ramadass Nagarajan, Karthikeyan Sankaralingam, Douglas C. Burger, Stephen W. Keckler, "A Design Space Evaluation of Grid Processor Architectures," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 40-51, Dec. 2001.

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

[173] Erik Nystrom, Ronald D. Barnes, Matthew C. Merten, and Wen-mei W. Hwu, "Code Reordering and Speculation Support for Dynamic Optimization Systems," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 163-174, Sep. 2001.

[174] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah Crandall, Erik Czernikowski, Leslie W. Jones IV, Diana Franklin, Venkatesh Akella, Frederic T. Chong, "Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor," *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 150-161, Jun. 2004.

[175] Subbarao Palacharla, Norman P. Jouppi, James E. Smith, "Complexity-Effective Superscalar Processors," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 206-218, Jun. 1997.

[176] Il Park, Michael D. Powell, T. N. Vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 171-182, Nov. 2002.

[177] Sanjay J. Patel, Daniel H. Friendly, Yale N. Patt, "Evaluation of Design Options for the Trace Cache Fetch Mechanism," *IEEE Transactions on Computers*, Vol. 48, No. 2, pp. 435-446, Feb. 1999.

[178] David A. Patterson, C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Journal of Solid-State Circuits, IEEE Transactions on Computers, Joint Special Issue on Microprocessors and Microcomputers*, Vol. 29, No. 2, pp. 108-116, Feb. 1980.

[179] Alex Peleg, Uri Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line," *US Patent 5,381,533*, 1994.

[180] Daniel G. Perez, Gilles Mouchard, Olivier Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," *Proceedings of the 2004 Workshop on Duplicating, Deconstructing and Debunking*, pp. 19-34, Jun. 2004.

[181] Karl Pettis, Robert C. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 16-27, Jun. 1990.

[182] Richard Phelan, "Improving ARM Code Density and Performance: New Thumb Extensions to the ARM Architecture," *ARM White Paper*, ARM Limited, Jun. 2003.

[183] Matthew Postiff, David Greene, Steven Raasch, Trevor Mudge, "Integrating Superscalar Processor Components to Implement Register Caching," *Proceedings of the International Conference on Supercomputing*, pp. 348-357, Jun. 2001.

[184] Alex Ramirez, Josep L. Larriba-Pey, Carlos Navarro, Josep Torrellas, Matero Valero, "Software Trace Cache," *Proceedings of the 13th International Conference on Supercomputing*, pp. 119-126, Jun. 1999.

[185] Alex Ramirez, Josep L. Larriba-Pey, Matero Valero, "Trace Cache Redundancy: Red & Blue Traces," *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp. 325-333, Jan. 2000.

[186] Alex Ramirez, Josep L. Larriba-Pey, Matero Valero, "The Effect of Code Reordering on Branch Prediction," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 189-198, Sep. 2000.

[187] Alex Ramirez, Oliverio J. Santana, Josep L. Larriba-Pey, Matero Valero, "Fetching Instruction Streams," *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 371-382, Nov. 2002.

[188] Stephen E. Raasch, Nathan L. Binkert, Steven K. Reinhardt, "A Scalable Instruction Queue Design Using Dependence Chains," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 318-329, Jun. 2002.

[189] Paul Racunas, Yale N. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitectures," *Proceedings of the 17th International Conference on Supercomputing*, pp. 22-31, Jun. 2003

[190] Ryan Rakvic, John P. Shen, "Parallel Cachelets," *Proceedings of the International Conference on Computer Design*, pp. 284-292, Sep. 2001.

[191] Glenn Reinman, Todd Austin, Brad Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 234-245, May 1999.

[192] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, John D. Owens, "Register Organization for Media Processing," *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp. 375-386, Jan. 2000.

[193] Alan Robinson, "Why Dynamic Translation?," *Transitive Technology White Paper*, Transitive Technologies, May 2001.

[194] Mendel Rosenblum, Edouard Bugnion, Scott Devine, Stephen A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, pp. 78-103, Jan. 1997.

[195] Roni Rosner, Avi Mendelson, Ronny Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 37-48, Sep. 2001.

[196] Eric Rotenberg, Steve Bennett, James E. Smith, "Trace Cache: A Low-Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, Dec. 1996.

[197] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, James E. Smith, "Trace Processors," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 138-148, Dec. 1997.

[198] Mariko Sakamoto, Akira Katsuno, Aiichiro Inoue, Takeo Asakawa, Haruhiko Ueno, Kuniki Morita, Yasunori Kimura, "Microarchitecture and Performance Analysis of a SPARC-V9 Microprocessor for Enterprise Server Systems," *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pp. 141-152, Feb. 2003.

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

[200] Subramanya S. Sastry, Rastislav Bodik, James E. Smith, "Rapid Profiling via Stratified Sampling," *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 278-289, Jun 2001.

[201] Yiannakis Sazeides, James E. Smith, "The Predictability of Data Values," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 248-258, Dec. 1997.

[202] Kevin Scott, Jack Davidson, "Strata: A Software Dynamic Translation Infrastructure," *Proceedings of the 3rd Workshop on Binary Translation*, Sep. 2001.

[203] Andre Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 295-306, Jun. 2002.

[204] Andre Seznec, Antony Fraboulet, "Effective Ahead Pipelining of Instruction Block Address Generation," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 241-252, Jun. 2003.

[205] Andre Seznec, Stephan Jordan, Pascal Sainrat, Pierre Michaud, "Multiple-Block Ahead Branch Predictors," *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 116-127, Oct. 1996.

[206] John P. Shen, Mikko H. Lipasti, "Modern Processor Design: Fundamentals of Superscalar Processors," *McGraw-Hill*, Jul. 2004.

[207] Gabriel M. Silberman, Kemal Ebcioglu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," *IEEE Computer*, Vol. 26, No. 6, pp. 39-56, Jun. 1993.

[208] Ronak Singhal, K. S. Venkatraman, Evan R. Cohn, John G. Holm, David A. Koufaty, Meng-Jang Lin, Mahesh J. Madhav, Markus Mattwandel, Nidhi Nidhi, Jonathan D. Pearce, Madhusudanan Seshadri, "Performance Analysis and Validation of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, Vol. 8, Issue 1, pp. 33-42, Feb. 2004.

[209] Michael Slater, "AMD's K5 Designed to Outrun Pentium," *Microprocessor Report*, Oct. 24, 1994.

[210] Michael Slater, "K6 to Boost AMD's Position in 1997," *Microprocessor Report*, Oct. 28, 1996.

[211] James E. Smith, Andrew R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, Vol. 37, No. 5, pp. 562-573, May 1988.

[212] James E. Smith, "Characterizing Computer Performance with a Single Number," *Communications of the ACM*, Vol. 31, No. 10, pp.1202-1206, Oct. 1988.

[213] James E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *IEEE Computer*, Vol. 22, No. 7, pp. 21-35, Jul. 1989.

[214] James E. Smith, Schlomo Weiss, "PowerPC 601 and Alpha 21064: A Tale of Two RISCs," *IEEE Computer*, Vol. 27, No. 6, pp. 46-58, Jun. 1994.

[215] James E. Smith, Gurindar S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, Vol. 83, No. 12, pp. 1609-1624, Dec. 1995.

[216] James E. Smith, "Instruction-Level Distributed Processing," *IEEE Computer*, Vol. 34, No. 4, pp. 59-65, Apr. 2001.

[217] James E. Smith, S. Subramaya Sastry, Timothy H. Heil, Todd M. Bezenek, "Achieving High Performance via Co-Designed Virtual Machines," *International Workshop on Innovative Architecture*, Maui High Performance Computer Center, Oct. 1998.

[218] Avinash Sodani, Gurindar S. Sohi, "An Empirical Study of Instruction Repetition," *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 35-45, Oct. 1998.

[219] Gurindar S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, Vol. 39, No. 3, pp. 349-359, Mar. 1990.

[220] Gurindar S. Sohi, Scott E. Breach, T. N. Vijaykumar, "Multiscalar processors," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 414-425, Jun. 1995.

[221] Gurindar S. Sohi, Amir Roth, "Speculative Multithreaded Processors," *IEEE Computer*, Vol. 34, No. 4, pp. 66-73, Apr. 2001.

[222] Eric Sprangle, Doug Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 25-34, Jun. 2002.

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

[224] Steven Swanson, Ken Michelson, Andrew Schwerin, Mark Oskin, "WaveScalar," *Proceedings of the 36th International Symposium on Microarchitecture*, pp. 291-302, Dec. 2003.

[225] Ariel Tamches, Barton P. Miller, "Dynamic Kernel I-Cache Optimization," *Proceedings of the 3rd Workshop on Binary Translation*, Sep. 2001.

[226] Thang Tran, David B. Witt, William M. Johnson, "Superscalar Microprocessor Including a High Speed Instruction Alignment Unit," *US Patent 5,991,869*, Nov. 1997.

[227] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, Anant Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 2-13, Jun. 2004

[228] Joel M. Tendler, J. Steve Dodson, J. S. Fields, Jr., Hung Le, Balaram Sinharoy, "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, Vol. 46, No. 1, pp.5-26, Jan. 2002.

[229] J. E. Thornton, "Design of a Computer, the Control Data 6600," *Scott, Foresman, and Co.*, Glenview, Illinois, 1970.

[230] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, Vol. 11, pp. 25-33, Jan. 1967.

[231] Jessica H. Tseng, Krste Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 62-71, Jun. 2003.

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

[233] Sriram Vajapeyam, Tulika Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 1-12, Jun. 1997.

[234] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, Anant Agarwal, "Baring It All to Software: Raw Machines," *IEEE Computer*, Vol. 30, No. 9, pp. 86-93, Sep. 1997.

[235] Steven Wallace, Nader Bagherzadeh, "A Scalable Register File Architecture for Dynamically Scheduled Processors," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 179-184, Oct. 1996.

[236] Shlomo Weiss, James E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 110-118, Jun. 1984.

[237] John Whaley, "Partial Method Compilation Using Dynamic Profile Information," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 166-179, Oct. 2001.

[238] Kenneth M. Wilson, Kunle Olukotun, "Designing High Bandwidth On-Chip Caches," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 121-132, Jun. 1997.

[239] Steven J. E. Wilton, Norman P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 5, pp. 677-688, May 1996.

[240] 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, May 1996.

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

[242] Adi Yoaz, Mattan Erez, Ronny Ronen, Stephen Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 42-53, Jun. 1999.

[243] Robert Yung, Neil C. Wilhelm, "Caching Processor General Registers," *Proceedings of the International Conference on Circuits Design*, pp. 307-312, Oct. 1995.

[244] Cindy Zheng, Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," *IEEE Computer*, Vol. 33, No. 3, pp. 47-53, Mar. 2000.

[245] Craig Zilles, Gurindar S. Sohi, "A Programmable Co-Processor for Profiling," *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pp.241-252, Jan. 2001.

# Appendix: Simulation Setup for Evaluating Control Transfer Support Mechanisms

In this appendix the identity translation framework used in Chapter 5 is described. In short, the framework is a stripped down version of the ILDP dynamic binary translation mechanism integrated with an earlier version of the baseline simulator described in Chapter 6.

The timing simulator part of the identity translation framework is largely based on the SimpleScalar 3.0C toolset [35] and is heavily modified (especially in the front-end) to closely model modern processor pipeline designs. The same timing simulator without the code caching facility is also used as a baseline for comparisons. This baseline configuration is referred to as *original*.

When program control flow reaches an existing superblock in the code cache, the simulator begins detailed timing simulation. Here the timing simulation starts with an initially empty pipeline. Similarly if an exit condition from the code cache is met (i.e., a superblock exit instruction's target is not currently cached in the code cache), the mode is changed to interpretation after the last instruction in the pipeline is committed. Overall performance is then measured as source instructions per cycle (IPC) for execution of all cached (and chained) instructions.

The superblock formation algorithm is the same as the one described in section 4.1.3. Similarly, a maximum superblock size of 200 instructions and a tail execution counter threshold of 50 were used.

To collect statistics, I use the SPEC CPU2000 integer benchmarks compiled for the Alpha EV6 ISA at the `base` optimization level (`–arch ev6 –non_shared –fast`). These are the same binaries used in Chapter 7. The `test` input set was used for all benchmarks except for

*253.perlbmk*, where one of the `train` input set (`-I./lib diffmail.pl 2 550 15 24 23 100`) was used. All benchmarks were run to completion or 4.3 billion instructions. NOPs defined in the Alpha ISA are properly recognized and removed when superblocks are formed. In *original*, NOPs are removed by the hardware in the decoding stage.

Figure A-1 shows the eight-stage out-of-order superscalar processor pipeline simulator used for the evaluation in section 5.3 (additional pipeline stages for the non-control transfer instructions such as the data cache related stages are not shown).

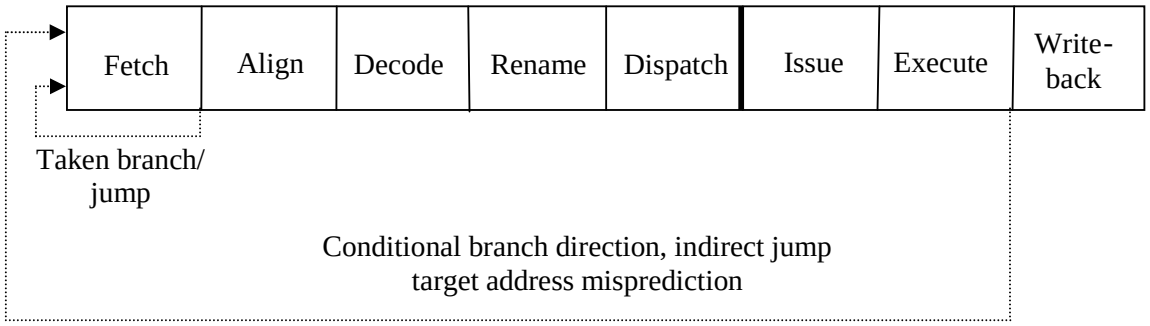



**Figure A-1 Simulated pipeline used in the identity translation framework**

Instruction fetch takes two cycles – one cycle for accessing the I-cache SRAM array and another cycle for shift and mask operation to select valid fetch target instructions. Control transfer information such as branch/jump types and their locations within an I-cache line is pre-decoded and stored in the I-cache array when the cache line is brought into the I-cache. All branch prediction mechanisms, i.e., branch (direction) predictor, BTB, and RAS were set to single-cycle. If the fetch redirection bubble cycles are accounted for (that is, if multi-cycle predictors are used), the performance differences in Figure 5-9 is further increased. Regarding predicted directions of conditional branches, when there is a disagreement between branch predictor and BTB for a conditional branch, the branch prediction overrides the BTB prediction. The out-of-order issue

mechanism in the pipeline backend and the cache and memory subsystems stay largely identical to the `sim-outorder` simulator. This led to higher IPC performance numbers in Chapter 5 compared with the evaluation using the baseline simulator in Chapter 7. Machine configurations are shown in Table A-1.

**Table A-1 Machine parameters used in the identity translation framework**

| | 4-way issue microarchitecture |
|---|---|
| Branch prediction | 16K-entry, 12-bit global history g-share branch predictor; 16-entry RAS; 2K-entry, 4-way set associative BTB; 256-entry fully associative, LRU-replacement JTLT (if used) |
| Branch predictor bandwidth | Up to 1 prediction per cycle |
| L1 I-cache | 32-KB size, direct-mapped, 64-byte line size, 2-cycle hit latency |
| Fetch bandwidth | Maximum 4 instructions per cycle;<br>Fetch continues after the first predicted not-taken conditional branch;<br>Fetch stops if a second branch is found |
| L1 D-cache | 32-KB size, 4-way set-associative, random replacement, 64-byte line size, 2-cycle hit latency, write-back, write-allocate |
| Unified L2 cache | 1-MB size, 4-way set-associative, random replacement, 128-byte line size, 8-cycle hit latency, write-back, write-allocate |
| Memory | 128-cycle latency, 64-bit wide, 4-cycle burst |
| Decode/issue/retire bandwidth | 4 |
| Issue window size | 64 |
| Execution resources | 4 integer units, 2 L1 D-cache ports, 2 floating-point adders, 2 floating-point multipliers |
| Reorder buffer size | 128 |