RUN-TIME OPTIMIZATION ARCHITECTURE

BY

MATTHEW CARL MERTEN

B.S., University of Illinois at Urbana-Champaign, 1996
M.S., University of Illinois at Urbana-Champaign, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

RUN-TIME OPTIMIZATION ARCHITECTURE

Matthew Carl Merten, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2002
Wen-mei W. Hwu, Adviser

Each new generation of wide-issue processors continues to achieve higher performance by exploiting greater amounts of instruction-level parallelism than the previous generation. Dynamic techniques such as out-of-order execution with hardware speculation have proven effective at increasing instruction throughput, parallelism, and utilization of processor resources. Run-time optimization techniques promise to enable an even higher level of performance by applying aggressive transformations at run-time that optimize across module boundaries, adapt code regions to changing input patterns, and customize code sequences for the underlying microarchitecture.

This thesis presents a hardware mechanism for generating and deploying run-time optimized code. The system exploits program execution phasing by automatically detecting and optimizing the instruction sequences that comprise the phase, called a hot spot. The hardware mechanism can be viewed as a filtering system that resides after the retirement stage of the processor pipeline, accepts an instruction execution stream as input, and produces instruction profiles and sets of linked, optimized traces as output. The code deployment mechanism uses an extension to the branch prediction mechanism to migrate execution into the new code without modifying the original code. These new components do not add delay to the execution of the program except during short bursts of reoptimization, because they operate in parallel with native execution. This technique provides a strong platform for run-time optimization because

the hot execution regions are extracted, optimized, and written to main memory for execution where they will persist across context switches. The framework is designed to preserve precise exception handling while applying optimizations which currently include partial function inlining (even into shared libraries), code straightening, loop unrolling, peephole optimizations, and instruction rescheduling with renaming, which are all concurrently performed with the running application.

To my wonderful wife, Polly.

ACKNOWLEDGMENTS

Throughout my graduate career at the University of Illinois, I have been fortunate to work with many talented people, more than I could ever adequately thank here. The high quality of students and professors in my classes, research group, and research lab has lead to fruitful discussions and ideas for which I am very grateful. But there are a handful of people that I do want to thank here personally who have made considerable contributions to this work and to my development as a researcher.

First, thanks are in order for my IMPACT Research Group predecessors whose technical knowledge and vision have produced some of the tools that we employ and the positive reputation that we currently enjoy. As our leader, Professor Wen-mei Hwu has guided our group on its research path, while allowing us the freedom to explore new ideas and concepts. He has freed us from funding concerns while providing the means for us to travel to numerous conferences, to have the latest equipment, and to generally focus on technical issues. During my senior year, Wen-mei brought me into the group as a fairly inexperienced undergraduate researcher. Through sometimes difficult years and disagreements, he has trained me to be a tougher, stronger, and more competent researcher. Yet I know that I still have a lot to learn. I greatly admire his ability to teach both through formal lesson and through humorous anecdote. He utilizes his ability to clearly distill complicated mechanisms down to the basic operations and simple examples to immeasurably help young students. I hope that I can replicate these talents in my own teaching. Likewise, I thank Professors Sanjay Patel, Sarita Adve, and Michael

Loui for being excellent teachers and for volunteering to help me through the final stages of my degree by serving on my doctoral committee.

In the countless hours of diligent and sometimes not-so-diligent work over the past six years, my colleagues have been a source of technical insight and personal friendship. Andy Trick, Chris George, and John Gyllenhaal made valuable contributions in the early stages of the ROAR project. In particular, Andy and I had many long discussions about program phasing and dynamic optimization in general. He is a particularly insightful engineer who I know will continue in excellence in all that he does. John served as a mentor and sounding board for both technical ideas and for interpersonal challenges, for which I am grateful. Erik Nystrom and Ronald Barnes have devoted considerable energy to the latter stages of ROAR, which may have inadvertently delayed their own master's theses. The three of us engineered a new execution-driven pipelined processor simulator complete with dynamic code optimization and management capabilities that provided a framework for our experimentation. Thanks to Chris Shannon, Andrew Schuh, and Professor Jose Nacho Navarro who have all assisted in ROAR infrastructure development. All of these colleagues have contributed in countless technical ways to ROAR that truly exemplify the phrase that our research team is "more than just the sum of its parts."

I would also like to recognize Hillery Hunter, who has contributed by being a constant support during both the difficult crunch times and the times of celebration. She is a woman that I greatly admire for her technical insight as well as her hard work, devotion, and faith. Our interaction has been able to transcend work life into a close friendship that I deeply appreciate. Thanks to John Sias for his vision and constant improvements to the IMPACT infrastructure.

While we have primarily worked on different projects, our friendship has helped us through challenging times. I'd also like to thank Richard Hank and Carol Thompson at Hewlett-Packard for the financial support and feedback on the ROAR project over the past four years.

My family has also been extremely supportive of my graduate school endeavors. My parents and parents-in-law have constantly encouraged me even when the research was not proceeding as hoped. They have built a bedrock of support by providing a solid education and values that have enabled me to work for my Ph.D. My wife, Polly, to whom this dissertation is dedicated, deserves unbounded thanks. She has often commented that she feels like she has gone through graduate school as well. Truly, she has enjoyed the ups and despaired in the downs along with me every step of the way. She has a great heart of love for those around her and has graciously committed to following me wherever my career takes me. In the song "Home to You," John Michael Montgomery declares:

> I get up and battle the day. Things don't always go my way.
> It might rain but that's okay. I get to come home to you.
> You are my best friend, And you are where my heart is,
> And I know at the day's end, I get to come home to you.

This sentiment I wish to resoundingly echo. I hope that I can serve and love her as well as she serves and loves me. Lastly, I thank my savior, Jesus Christ, for the faith, love, and hope that He instills in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Out-of-order execution and automated dynamic speculation have dramatically improved the performance of modern microprocessors. These techniques were the first steps toward allowing the microprocessor *itself* to determine how to execute code efficiently. Thus far, such hardware optimizations have been limited in scope and have typically been made on-the-fly without any persistent record. More aggressive and persistent optimizations have instead been accomplished in software through the use of optimizing compilers. However, many of these optimizations rely on accurate profile information to profitably transform code. While compilers often support the use of profile information, software vendors have been reluctant to add the profiling step to their development cycles. Not only is the determination of a representative profile difficult, the behavior of certain programs may change due to time or space overhead during profiling. For these reasons, an automated, transparent mechanism for profiling and reoptimizing code based on current usage would be advantageous. Furthermore, a dynamic system could improve performance in ways that a static optimizer cannot. As program behavior changes over time, a dynamic system could reoptimize code to take advantage of temporal relationships. While a typical static compiler optimizes for the average behavior across the entire execution, a dynamic system could lead to more directed optimizations. In addition,

since optimization is performed on the same machine that runs the application, the optimizer can specifically target that system.

1.1 Run-Time Optimization Architecture Principles

Hardware support for dynamic profiling and reoptimization reduces the reliance on software systems to perform these tasks. The use of dedicated, background hardware mechanisms allows for limited run-time overhead. The proposed hardware system, called Run-Time Optimization Architecture (ROAR), is designed to automatically and transparently profile applications while generating sets of traces that cover the frequently executed paths of an application. In parallel with program execution, these traces are then fed to a hardware optimizer that improves code quality through the application of optimizations and instruction speculation and scheduling. Located after the retirement stage of the processor pipeline, the proposed system is designed to incur negligible overhead and minimally affect the design of the pipeline.

The profiling and extraction mechanism can be viewed as a filtering system that accepts an instruction execution stream as input and produces instruction profiles and sets of traces as output. As shown in Figure 1.1, the conceptual filter consists of three primary components responsible for collecting profile information (*Branch Behavior Buffer*), determining the execution coverage of the profiled instructions (*Hot Spot Detection Counter*), and generating traces for the frequently executed paths (*Trace Generation Unit*).

During application execution, the Branch Behavior Buffer component determines the most frequently executed branch instructions while collecting a profile of their behavior. The buffer

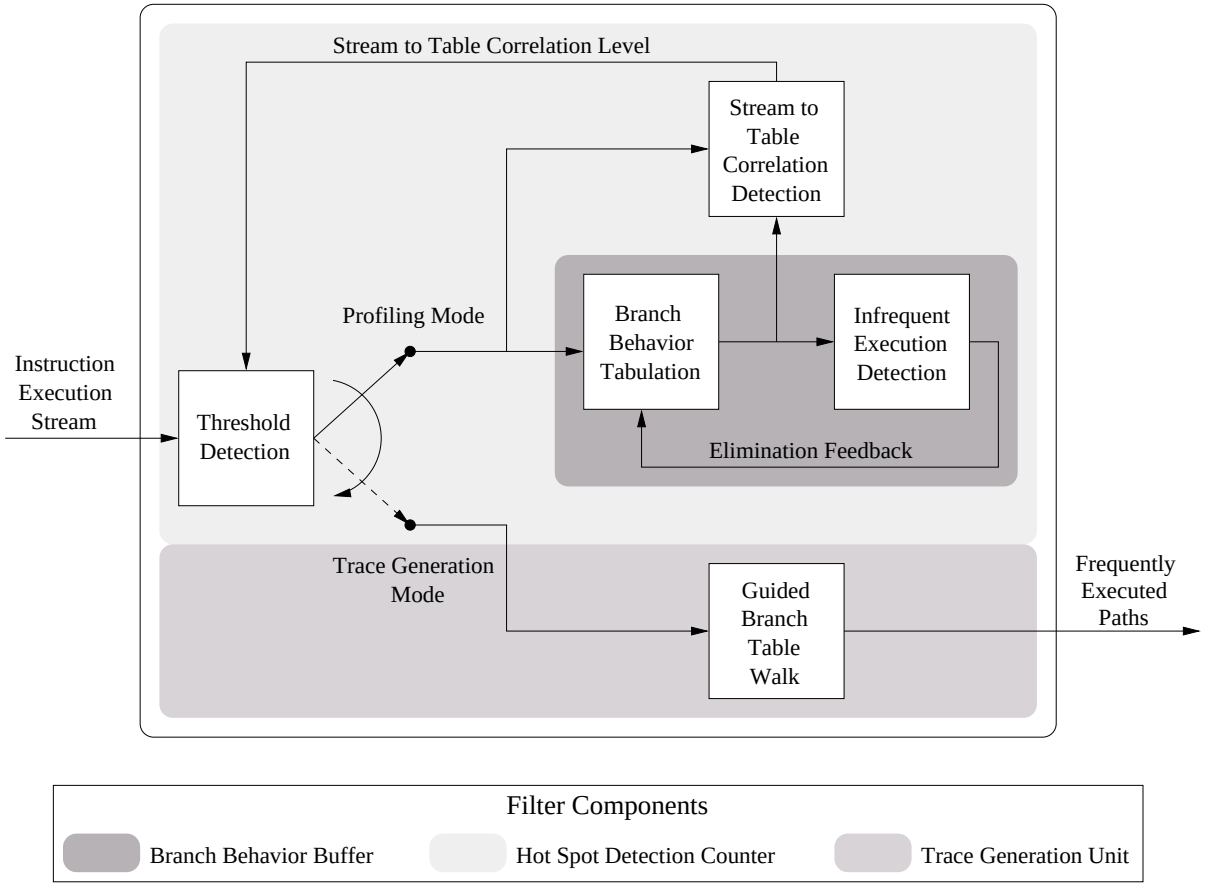


Figure 1.1 Hot Spot Detector and Trace Generation Unit depicted as an on-line filter.

consists of a hardware table used to collect the profiles of executed branches through the *Branch Behavior Tabulation* process. Each entry consists of a branch identifier, profile counter, and several other bookkeeping fields. The *Infrequent Execution Detection* process eliminates infrequently executed branches from the table to filter out those that only spuriously execute. Meanwhile, the Hot Spot Detection Counter component is responsible for determining when execution is primarily confined to the collected branches. The *Stream to Table Correlation Detection* process compares the execution stream to entries in the table and forces a transition to *Trace Generation Mode* from *Profile Mode* when a comparison threshold is met. The set of

branches in the table when the transition occurs are the most frequently executed branches for the current pattern of execution. The branches are referred to as *hot spot branches* which serve as the skeleton of the region of code, called a *hot spot region*, or simply *hot spot*, that comprises this pattern. Because the detection process is performed quickly at run time, the region can be constructed and optimized during execution while leaving sufficient time to benefit from execution in the newly optimized code. Program hot spots provide an excellent opportunity for run-time optimization because they can be quickly identified and contain only the frequently executed code. Chapter 3 provides evidence that programs often exhibit phased hot spot execution behavior. The Branch Behavior Buffer and Hot Spot Detection Counter components are collectively referred to as the *Hot Spot Detector* [1], [2] and are more thoroughly discussed in Chapter 4.

Using the profiles gathered by the Hot Spot Detector, a hardware component called the Trace Generation Unit [2], [3] produces sets of traces that cover the frequently executed paths in the code. The *Guided Branch Table Walk* process utilizes the execution stream to walk the hot spot branch skeleton stored in the branch table to produce the traces. The use of the stream to step through the table entries eliminates the need to store all instructions during the profile phase and prevents inclusion of entries in the table that were short-lived and are no longer active. Meanwhile, the use of the skeleton ensures that the traces are representative of the frequently executed instructions. The instructions in the output traces are generated in execution order, thus providing an inherent code-straightening optimization. The traces are linked together by their exit branches to provide a means for extensive execution within the optimized hot spot. The output of the detector can be used to feed and direct a hardware-

based reoptimization mechanism, or it can serve as a profiling platform for a software-based reoptimizer. One direct application of this mechanism is to perform code optimizations that improve instruction fetch performance such as loop unrolling, partial function inlining, and branch promotion. The output traces are stored in a *memory-based code cache*, thus enabling a traditional instruction cache to fetch multiple basic blocks per cycle, and preserving optimized hot spots for prolonged execution. The Trace Generation Unit is discussed more thoroughly in Chapter 5.

Besides the optimizations performed by the Trace Generation Unit, additional improvements can be made to further enhance trace execution speed. Instruction scheduling, for example, that targets the specific, local microarchitecture greatly improves application performance through increased utilization of processor resources and accurate timing of dependent instructions. While an out-of-order execution engine performs instruction scheduling for each dynamic instruction, it requires significant complexity in the front end of the processor pipeline, and it operates continuously. The wake-up, select, and rename logic in the front end are atomic operations (not easily pipelined) that are among the components with the longest critical path delay [4]. By scheduling the static instructions in the frequently executed traces, the scheduling hardware can be moved off the critical path in the pipeline, thus reducing pipeline complexity. Furthermore, the scheduling costs can be amortized over the dynamic executions of the trace, thus allowing more time to make better scheduling choices. While this approach may not achieve all of the benefits of scheduling for each dynamic trace, it does provide a platform for performing additional code restructuring optimizations. The optimizer and scheduler are discussed in depth in Chapter 6.

1.2 Contributions

In this dissertation, I show that run-time optimization is feasible and beneficial by exploiting program phasing behavior through hot spot optimization. Initial prototypes of the necessary components and algorithms are presented that describe the system. The primary components include a low-overhead profiler, an instruction trace formation engine, an instruction scheduler that utilizes a new speculation method, and a code deployment and management mechanism. In addition, an interface to compiler-generated code analysis data is also utilized to enable more aggressive instruction scheduling. Experimental results are presented to quantify the effectiveness of the overall system as well as the individual components.

I also show that these optimization techniques can be transparently applied to the application. The application binary image can be preserved for correct data accesses into the code segment. For example, applications that perform self-checks will still pass their checks, and applications that modify their own code will still execute correctly. Likewise, precise exception handling after optimization must be preserved. Speculation techniques and related system components designed to make these guarantees are prototyped and analyzed to ensure transparency. Issues concerning system operation, optimization, and monitoring are discussed to provide insight into the source of the benefit.

CHAPTER 2

DYNAMIC OPTIMIZATION

In the early years of computer programming, the highest application performance was usually achieved by hand coding routines in assembly language. This method, however, often proved time consuming and error prone. To improve programmer efficiency, high-level languages were introduced to speed the application development processes by automating and broadening the scope of many development steps. Compilers are now relied upon to translate the high-level language representation of an application into efficient assembly routines. To accomplish this task, the compiler applies various *optimizations* to the application code in an effort to improve its performance. While a standard term in the compiler community, the term *optimization* is actually a misnomer since compilers do not attempt to find the single optimally fastest set of instructions to perform a particular task. Such an optimal set is highly dependent on the consistency of and patterns in the input data; an application that only executes on a single instance of input values is likely to consist of a set of instructions that simply generate the desired output. Rather, each optimization attempts to improve overall application performance in relation the expected variety of inputs [5]. Therefore, an optimization may favor a particular input pattern, resulting in improved performance for that pattern at the expense of other patterns.

While typically limited to the compiler, optimizations may be applied to an application at various points during its lifetime, from development to deployment. *Post-link* optimization is a class of techniques for optimizing binary and library components after they have left the software house. The terms *dynamic optimization* and *run-time optimization* imply the process of altering the application's instruction sequences in response to particular behaviors or inputs, and generally refer to alterations made while the software is in use. Section 2.1 summarizes and describes the various occasions for optimization during an application's lifetime, Section 2.2 describes situations that a post-link optimizer might exploit, and Section 2.3 provides an overview of the proposed processor architecture that exploits dynamic opportunities.

2.1 Occasions for Optimization

Figure 2.1 presents a generalized characterization of the amount of available information during various stages of software development and deployment. Early in the compilation process, extensive program analysis is employed to provide in-depth understanding of relationships within an application module. However, little information about the eventual execution environment, such as available processor resources or other interconnecting modules, is typically known. Modern compilers do use *assumed* environment information, such as control-flow profiles or architecture descriptions, so that aggressive optimizations can be applied. During execution of the compiled application, true assumptions will yield high performance, but poor assumptions may yield worse performance than without the optimizations at all. For example, the compiler determines the instruction schedule based upon assumed latency and resource

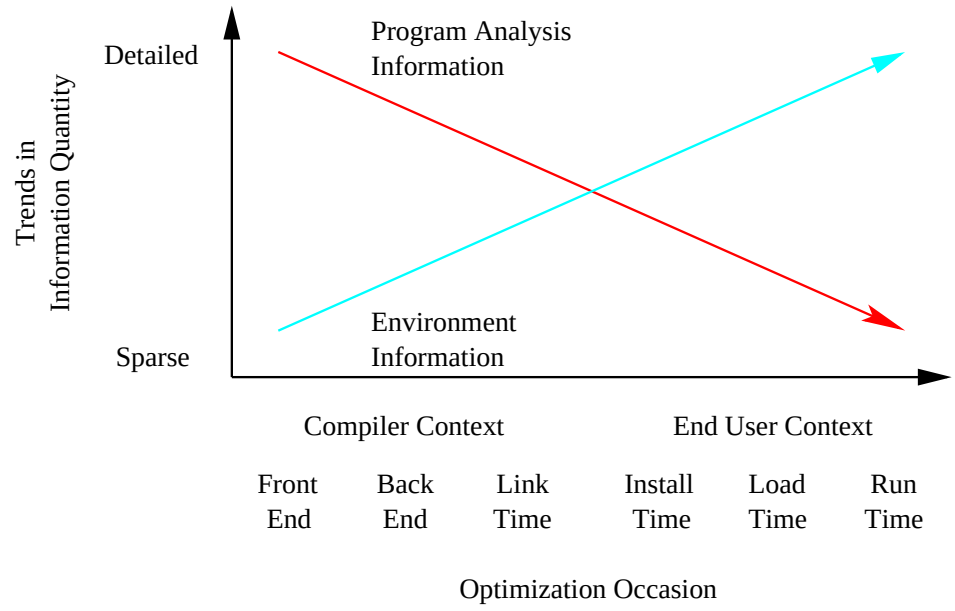


Figure 2.1 Trends in the quantity of available information during various stages of software development and deployment.

availability. But if the architecture requires a longer latency for a particular instruction or has more functional units than assumed, the processor may unnecessarily stall or under-utilize the available resources, respectively.

Once the application arrives in the user's local environment, much of the voluminous program analysis information has been discarded by the compiler and linker, but the particulars of the exact environment begin to crystallize. The features of the processor become known when the application is installed, exact libraries become known when they are loaded, and exact usage patterns become known when the application is executed. Even though a great wealth of information pertinent to efficient execution has been gained, most program analysis information has been lost. Only the very instructions themselves are presented to the execution core, leaving little means for program adaptation to the new environmental information.

Three primary options exist to correct this imbalance:

1. Additional environmental information can be provided to the compiler. Modern compilers utilize detailed machine descriptions and control-flow profiles to carefully guide optimization and scheduling. But all of this information is simply a guess as to what the future environment will be.

2. The run-time system can infer and speculate as to what information the analyses generated. Modern out-of-order processors may use predictors to speculate which loads and stores do not alias so they can be reordered on-the-fly. However, a recovery mechanism is required in case the speculation was incorrect.

3. The compiler can annotate modules with analysis information, thus shifting some of the compilation responsibility to the load-time and run-time systems when the actual environmental information is available. Compiler alias analysis information could be preserved in the form of instruction annotations or in a processor-accessible table. For example, the information might specify which loads and stores can be safely reordered. Alternatively, the compiler could generate an intermediate representation for distribution, relying on the run-time system to perform extensive further compilation in the local environment.

As previously alluded to, modern application development and deployment practices generally only employ code optimization at compile time. Software vendors compile program source at the software house and distribute application codes to users in the form of binaries and libraries, as is shown at the top of Figure 2.2. Users then install the application in their local

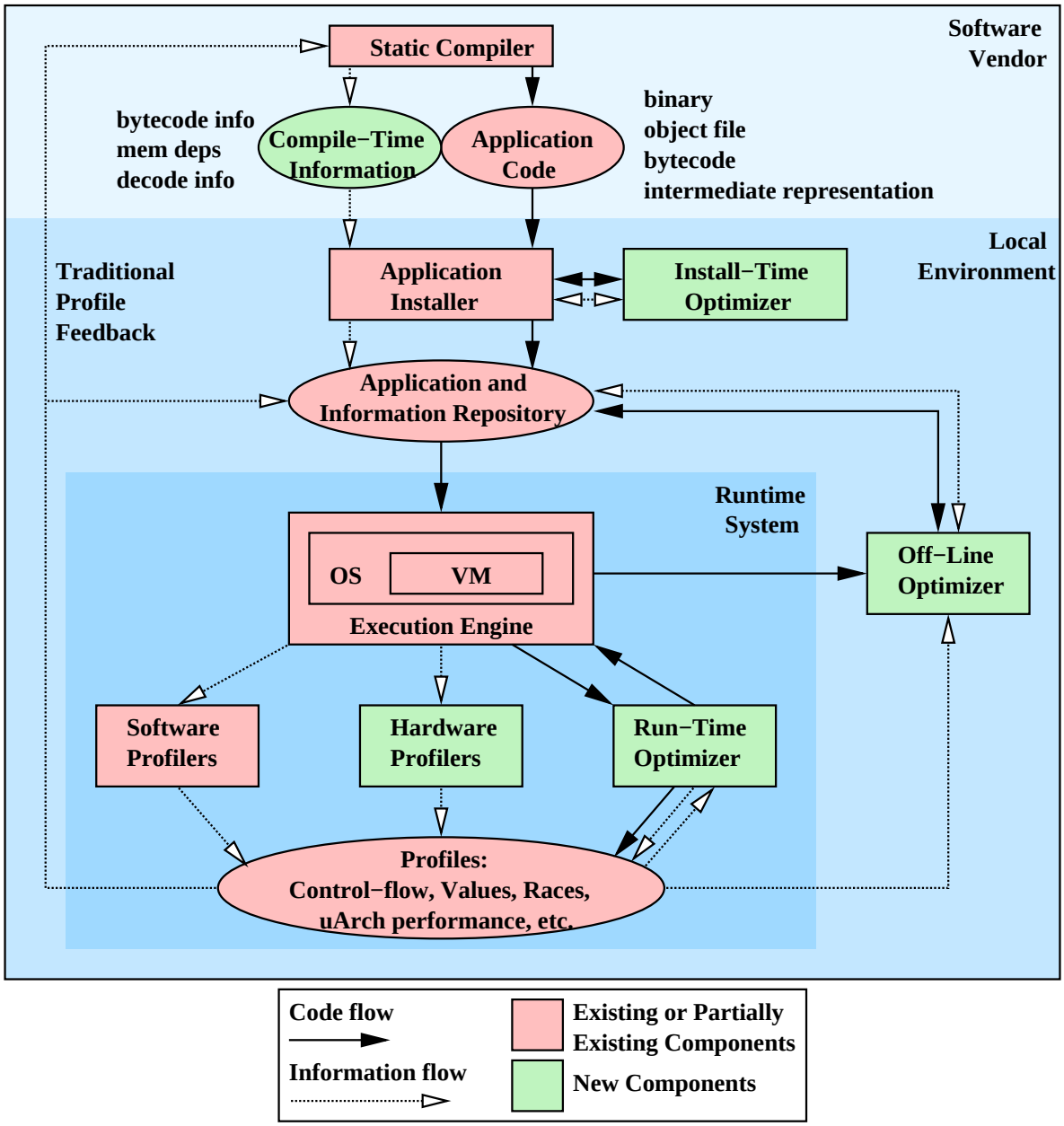


Figure 2.2 A comprehensive post-link optimization system.

environment into a *repository* (which may be simply the file system along with some locator information in the Windows Registry [6], for example). When a user invokes an application, its components are loaded, dynamically linked, and executed almost exclusively without modification.

These traditional practices leave little opportunity for an application to adapt to a new environment or input pattern that differs from the software vendor's assumptions. Some modern systems rely on dynamic code generation or self-modifying code for program adaptation, but these often result in considerable overhead or limited applicability, respectively. Others rely on various forms of post-link optimization to perform the adaptations. Figure 2.2 depicts several different post-link mechanisms for enabling dynamic adaptation. During the installation process, the processor microarchitecture becomes known, thus enabling *install-time* customization for various processor features, resources, and latencies. Similarly, optimization during processor idle time, commonly referred to as *off-line* optimization, may best take advantage of common usage, data, or input patterns. Install-time optimization opportunities may simply be a subset of off-line opportunities and, therefore, the install-time and off-line optimizers could be the same system component. *Run-time* optimization has the potential to adapt codes on-the-fly for actual current usage, data, and input patterns, even if they conflict with the common or average patterns, thus providing a unique ability for fine-grained adaptation. Both the off-line and run-time optimizers depend upon profilers (hardware, software, or a combination thereof) that gather short-term and long-term usage, data, and input profiles. They are shown inside the Runtime System in Figure 2.2.

A wide variety of code optimization opportunities are available to a post-link optimizer. Many optimizations have degrees of aggressiveness ranging from simple, block-level strength to comprehensive, program-level strength. For example, predication within code regions may range from basic if-then-else conditional if-conversion to aggressive whole-function if-conversion with predicate logic reduction. While the levels of aggressiveness are usually continua of strengths, *light-weight* and *heavy-weight* are often used to describe the two extremes.

Because the cost of optimization directly reduces any benefit from the optimizations in run-time optimization systems, the overhead of the algorithms must be minimal. Typically, overhead prohibits broad program analysis because of the time required to iterate over the code and because of the space required to contain the state of the analysis. Optimizations that have limited scope over the code and whose state can be contained in limited space can generally be referred to as light-weight optimizations. They are expected to produce reasonable results at low cost. To mitigate some of the time and space cost, run-time optimization can benefit from the use of dedicated hardware structures. Off-line reoptimization systems whose time and space constraints are often considerably less stringent are likely to employ more heavy-weight optimizations. They are expected to produce high-quality results but can tolerate much higher costs. Off-line systems may employ some of the same dedicated hardware structures as run-time optimizers to reduce the expense of gathering profile information. A comparison of the approaches is presented in Table 2.1.

Table 2.1 Comparison of run-time to off-line optimization.

	Run-time	Off-line
Aggressiveness	Moderate (light-weight)	Aggressive (heavy-weight)
Speed	Swift	Deliberate
Scope	Localized	Wide-spread
Focus	Single existing situation	All likely situations
Program analysis	Via hints or easily accessible stored information	Extensive analysis or stored information
Optimization system	Heavy reliance on hardware	Software with potential for hardware assistance
Example: predication	Simple merging and stripping of if-then-else paths	Full if-conversion, logic reduction, and partial reverse if-conversion

2.2 Optimization Opportunities

Post-link optimization opportunities abound as greater information about the use and environment of the application become known. Several general classifications can summarize the post-link optimization opportunities:

Software boundaries. Modern software engineering techniques often lead to modular programs that have been written as a collection of components that are linked together at run time. During compilation of a particular module, information about the others is unknown. Many production compilers today have little knowledge of other source files in the same module, let alone other modules. This trend continues with new distributed/web-based applications: Microsoft's .NET [7], IBM's Websphere [8], HP's E-Services [9], etc. Software boundaries are also encountered when considering layered software, for example, where a user application may communicate through a runtime system to a Web server, which then communicates through the operating system into a device driver.

Actual environment. Modern compilers typically generate code for the lowest-common-denominator machine model that the code is expected to run on, although occasionally customized code sequences for particular tasks are specialized for various machine models. A post-link optimizer could adapt the code for the particular instruction set characteristics (i.e., instruction and functional unit availability, instruction latencies). Further optimizations may be made when the characteristics of the caches (size and replacement policies) and branch prediction mechanism are known.

Phased execution behavior. Many programs have common pieces of code that are used in different ways with different control-flow patterns at different times during execution. A post-link optimizer could perform on-line profiling to detect the various usage patterns and then generate a custom piece of optimized code for each pattern.

Input stimuli. The user input and data input sets often control the tasks performed by the application. Control flow and data value patterns could be exploited by an optimizer.

Compiler analysis shortfalls. Selective profiling may be employed to uncover desirable but unprovable relationships. Consider a compiler pointer analysis algorithm that determines that two pointers may alias to the same location. In reality, they may never actually alias, but profiling is required to determine this relationship. With such information, data speculation could be used to reorder memory accesses using the two pointers.

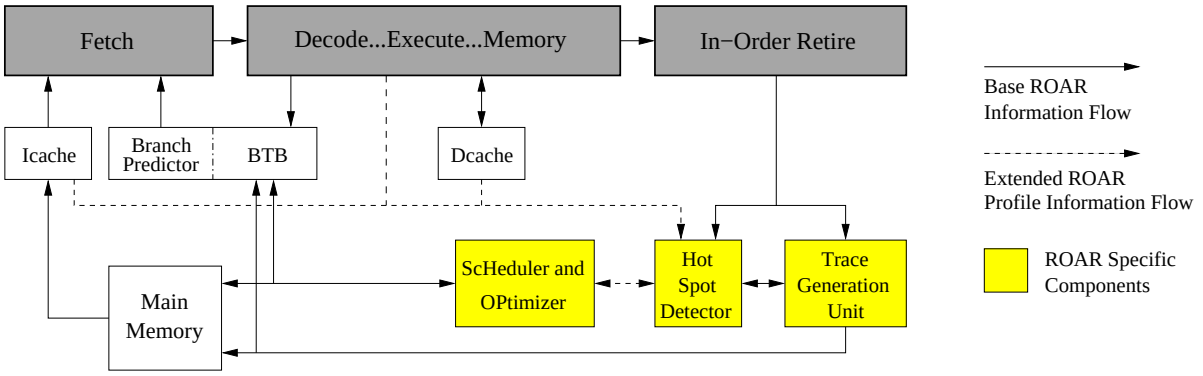


Figure 2.3 Instruction supply and retirement portions of the processor with highlighted hot spot detection and trace generation hardware.

2.3 Run-Time Optimization Architecture Overview

The proposed Run-Time Optimization Architecture utilizes aspects of options 2 and 3 presented early in Section 2.1, in a combined framework, where run-time optimization is performed using both generated and stored analysis information. For example, the architecture relies on hardware profiling techniques to identify important code regions and uses stored memory dependence information to aid instruction rescheduling. Use of ROAR also assumes that the compiler may perform aggressive optimizations based on assumed information, option 1. This approach utilizes the compiler to assist in finding and exposing instruction-level parallelism (ILP) while allowing ROAR to focus on situations where the compiler made faulty assumptions, had insufficient scope, or was only able to optimize for average or compile-time profiled behaviors.

The ROAR architecture features a number of components designed to enable efficient dynamic adaptation in processors. Shown in Figure 2.3, these components are located after the retirement stage of the pipeline, off the critical path of the processor core, and are designed to

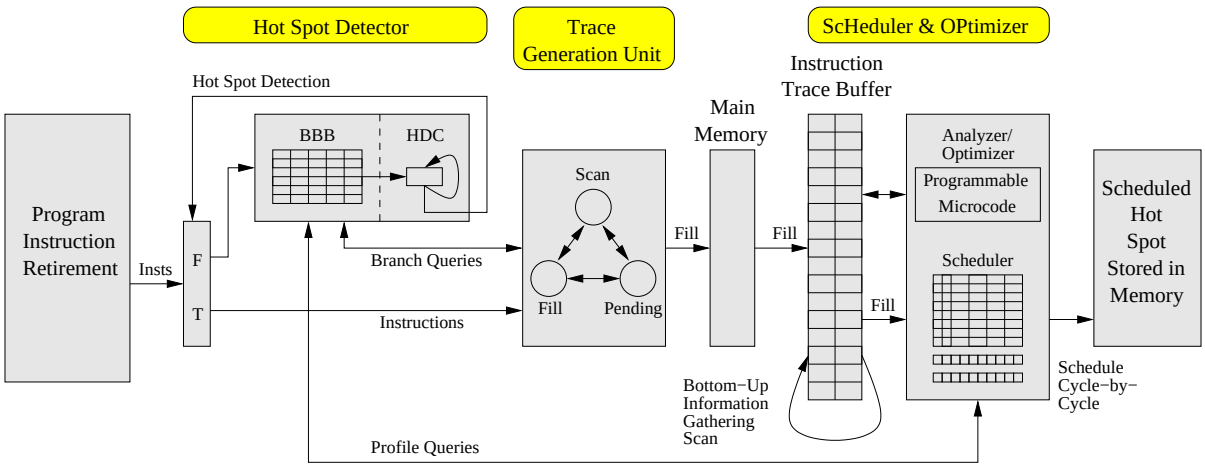


Figure 2.4 Optimization process flow.

make swift and focused—in essence, surgical—alterations to executing code. For Explicitly Parallel Instruction Computing (EPIC) processors [10], initial execution relies on the *plan of execution* (POE), or explicit instruction schedule, as specified by the compiler. During execution, a monitoring component coupled with an advanced profiler, called the Hot Spot Detector, tracks executed instructions after the retirement stage of the pipeline for optimization opportunities, as shown in Figure 2.4. The profiler seeks to detect program execution phasing, a behavior typical of many applications. For each phase, it collects information that will eventually lead to the identification of the static instructions that comprise the phase, called the hot spot. Specifically, the profiler identifies the frequently executed branch instructions within the phase, called the hot spot branches.

Once an opportunity is detected, the Trace Generation Unit is invoked to extract key executed traces in the hot spot for further dynamic optimization. The Trace Generation Unit tracks the retirement of subsequent instructions, utilizing the retirement stream to construct traces that

fall within the bounds of the detected hot spot. These generated traces collectively contain a vast majority of the frequently executed instructions for a particular phase of program execution. These traces are stored in a memory-based code cache for subsequent execution and management. Because the traces are stored in memory, the traditional pipeline fetch mechanism can be utilized to funnel the traces into the processor. Entry points from original code into the code cache are maintained in an enhanced branch target buffer [2] which redirects execution when transition points are encountered.

Once the traces for a hot spot have been formed, they are streamed into a hardware instruction Scheduler and Optimizer, called the SHOP, to form a new POE tuned and customized for the current execution conditions. These mechanisms guarantee that precise exceptions are preserved by utilizing an enhanced form of sentinel speculation, called *Precise Speculation* [11], which allows for an explicit representation of reordered instructions. Generation of the new POE is aided by memory dependence analysis information which was annotated on the memory operations at compile-time. Finally, the new POE is written into the code cache for extended execution, persisting through multiple context switches.

CHAPTER 3

PROGRAM CHARACTERISTICS

Many applications exhibit behavior conducive to run-time profiling and optimization. For example, program execution often occurs in distinct phases, where each phase consists of a set of code blocks that are executed with a high degree of temporal locality. When a collection of intensively executed blocks also has a small static footprint, a highly favorable opportunity for run-time optimization exists. As previously defined, such a collection of blocks comprise a region of code that is a hot spot. Commonly, the term hot spot is often defined in relation to the rule-of-thumb that states that 90% of the dynamic execution is spent in less than 10% of the static code. This definition suggests that the 10% of the static code is comprised of tight loops called hot spots. While hot spots are loops, they are actually often found to have a more sophisticated structure; they are often comprised of an outer loop that may contain several inner loops and complex control flow that may span multiple functions. A run-time optimizer can take advantage of execution phases by isolating and optimizing the corresponding hot spot instruction blocks present throughout an application. Ideally, aggressively optimized code would be deployed early in each phase and exercised heavily throughout each phase until the execution patterns shift. To maintain control over the mass of dynamically optimized code, optimized hot spots that are no longer active may be discarded, if necessary, to reclaim memory space for newly optimized code. They may also be saved for later use for recurring phases.

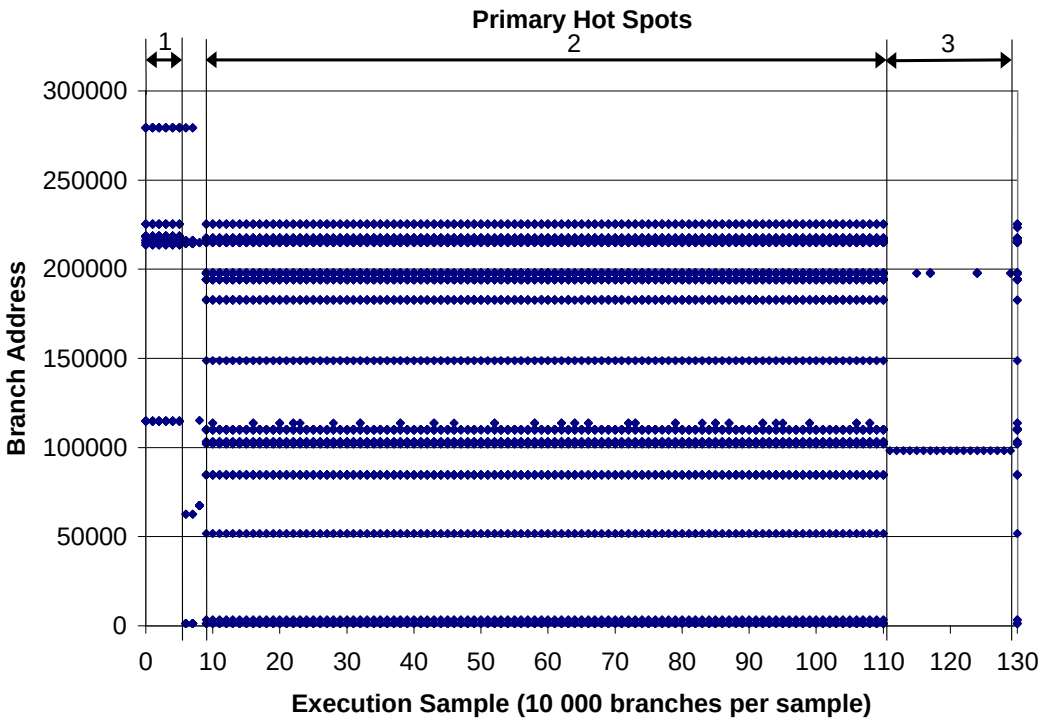


Figure 3.1 Important branches executed in each execution sample for *134.perl*. Each data point represents a branch that executed at least 40 times within the sample duration of 10 000 branches. These samples of 10 000 contiguous branches are taken once every 2 million branches.

3.1 Program Phasing

An example of application phasing behavior can be seen in a Perl interpreter (*134.perl* from the SPEC CPU95 [12] benchmark suite) running a word jumble script. As shown in Figure 3.1, this application contains three primary, distinct phases of execution with one hot spot per phase. Hot spot 1 runs for 72 million instructions, hot spot 2 for 1.35 billion, and hot spot 3 for 200 million. The first hot spot involves reading in a dictionary and storing it in an internal data structure. The second hot spot processes each word in the dictionary, and the

third scrambles a selected set of words in the dictionary. Figure 3.1 also indicates that, within each hot spot, the addresses of the intensely executed instructions are not typically clustered in one small address range, but instead have components that are widely scattered throughout the program. Hot Spots 1 and 2 share some common code, namely the command invocation and expression evaluation functions, that clearly call different routines based on the particular phase. Hot Spots 2 and 3 also share some code, just below address `200000`, but it is executed near the frequency threshold in Hot Spot 3.

The second hot spot serves as an excellent example of why run-time optimization is needed. The input script exercises Perl's `split()` routine, which breaks up an input word into individual letters, and `sort()`, which sorts those letters. The first function, `split()`, calls a complicated regular expression matching algorithm with a simple, null regular expression pattern (which splits the string into separate characters) [13]. Because execution consistently traverses a small number of paths within the functions that comprise the algorithm, this region of code would benefit from partial inlining and code layout, followed by path optimization. A static compiler could perform these optimizations, but the larger code size and compile time would be wasted for most input scripts. The second function, `sort()`, calls the library function `qsort()`, which then calls a Perl-specific comparison function, which in turn calls the library function `memcmp()`. Less than half of the code in the comparison function is ever executed because only strings of single characters are actually sorted. Function inlining is clearly also an effective optimization of the region in this example because of the frequent cross-module calls to the library comparison function. However, an optimizer that operates inside the user's environment is needed to support inlining across library and application boundaries. In addition, if

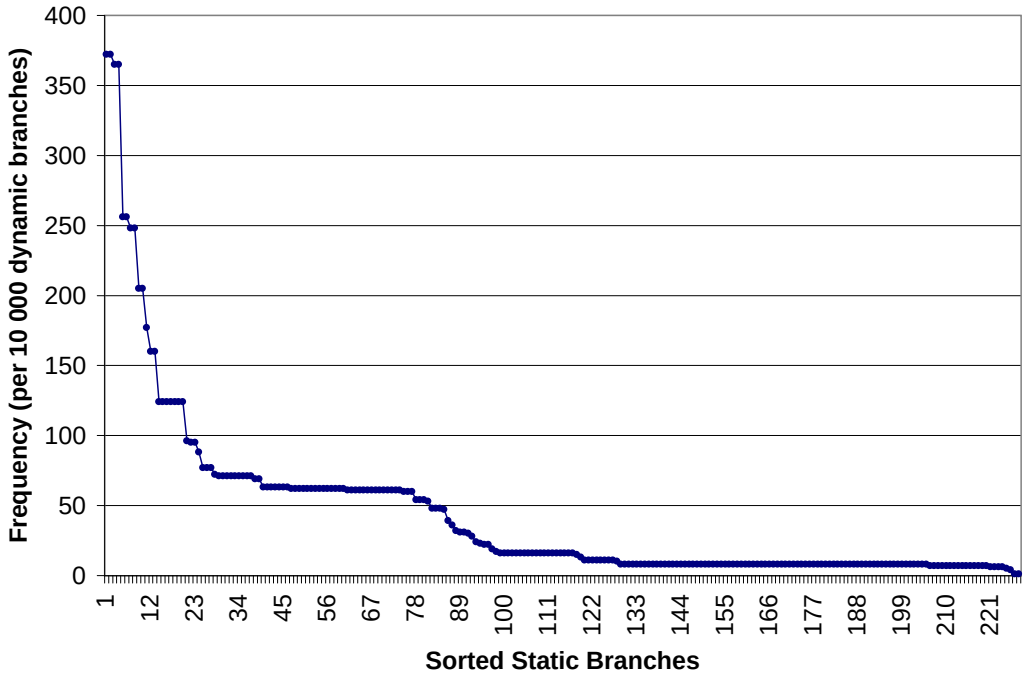


Figure 3.2 Profile distribution for the second primary hot spot in *134.perl*. Branches sorted from most to least frequent.

dynamically linked libraries are used, inlining must be delayed until the application is loaded or run. Phase-based optimization is also beneficial in order to optimize the common functions for the phase's specific activity. In this example, both the common evaluation functions along with the string library functions can be tailored and inlined. Phase-based optimization is difficult for a compiler because it typically cannot detect phase-specific behavior due to aggregate profiling, nor can it effectively generate different versions of functions without having differentiated call sites.

The *working set* of a phase is defined as all of the static instructions executed during the phase. Figure 3.2 shows a branch profile of the working set for a typical 10 000 branch sample

of Perl's second hot spot. The data reveals that a subset of the static branches in the working set account for the vast majority of the dynamic instances in the sample (for example, branches 0 through 122). These intensely executed branches define the boundaries of the hot spot region, and effective optimization can be limited to that portion of the working set. By including infrequent branches into the optimized version, the optimization costs could potentially exceed the marginal improvement gained. A number of strategies could be employed to determine which branches to include and which to leave out, including a simple heuristically chosen minimum frequency.

Through inlining and optimization, compilers often take partial advantage of program phasing. By using traditional profiling techniques, heavily executed call sites can be identified and the callee inlined into the caller. Compiler inlining has the tendency to gather tightly coupled pieces of code together, sometimes bringing the components of a phase together. However, the traditional compilation and deployment model has a number of drawbacks to effective inlining. First, while components may be inlined into many caller locations, the profile for the component is generally just its average profile over the course of the whole execution. Even though the component may behave quite differently in its various contexts, the compiler will use the average profile during optimization and code layout, missing customization opportunities. A second round of profiling or a more comprehensive whole program path [14] first round of profiling could be employed, but the software industry typically has been reluctant to perform even a single round of simple edge-weight profiling. Second, the trend in software distribution is toward more modular applications. This tendency limits the amount of inlining and inter-module optimization that can be performed by the compiler because its scope is limited to a

single module. Hence, run-time optimization has the potential to be an even more important technique in future run-time systems.

3.2 Hot Spot Characteristics

Programs often contain some functions that appear in multiple hot spots. For instance, this commonly occurs with internal library functions that are called from different locations within the program. Naturally, the behavior of these functions may vary depending on the *calling context*, which is defined as the dynamic path through the call graph taken to reach a given function. One such example is the function `str_new()` from the Perl interpreter, shown in Figure 3.3. This function is present in the three main hot spots and has three different profiles, each of which were gathered by the mechanism described in subsequent chapters. Figures 3.4(a)-(c) depict these three versions and are annotated by profile weights collected during a short window early in each phase (by the Hot Spot Detector). The dark arrows and blocks indicate the important edges and basic blocks as determined by the profile weights. The common paths through the function differ for each of the three execution phases. Note that branch `x867` from block `E` is missing from the profile for Hot Spot 3. This situation results from contention for resources within the hardware detector. For blocks that end in a branch, the block weight is the branch execution weight, while for other blocks, the weight is derived from the known input arcs.

The profile of Hot Spot 1 reveals that branch `x829` in block `A` always branches to block `C`. Branch `x829` decides whether a previously freed string is available from the string free list, or

```
             STR *
             str_new(len)
             STRLEN len;
             {
A:               register STR *str;

   0x421820      if (freestrroot) {
B:                   str = freestrroot;
                     freestrroot = str->str_magic;
                     str->str_magic = Nullstr;
                     str->str_state = SS_NORM;
   0x42183E          // direct jump to D: if (len) block
                 }
                 else {
C:                   Newz(700+x,str,1,STR);
                     // Newz macro(x,v,n,t):
                     //   (v = (t*)safemalloc((MEM_SIZE)((n)
                     //                       * sizeof(t))))
                     //   memzero((char*)(v), (n) * sizeof(t))
                 }
D: 0x42185F      if (len)
                     STR_GROW(str, len + 1);
                     // STR_GROW macro (str,len):
E: 0x421867          // if ((str)->str_len < (len))
F:                   //     str_grow(str,len)
G:               return str;
             }
```

Figure 3.3 Source code for function `str_new()` in the application *134.perl* compiled for x86 annotated with the branch addresses.

whether a new string must be created. For Hot Spot 1, the free string list is empty each time the function is called. This is consistent with the Perl input script which begins by reading a dictionary file and creating new strings for the words. This operation requires no string deletions and hence no strings are added to the free list. The opposite is true for Hot Spot 2, where a free string is always available from the free list. Clearly, phase-based customization of

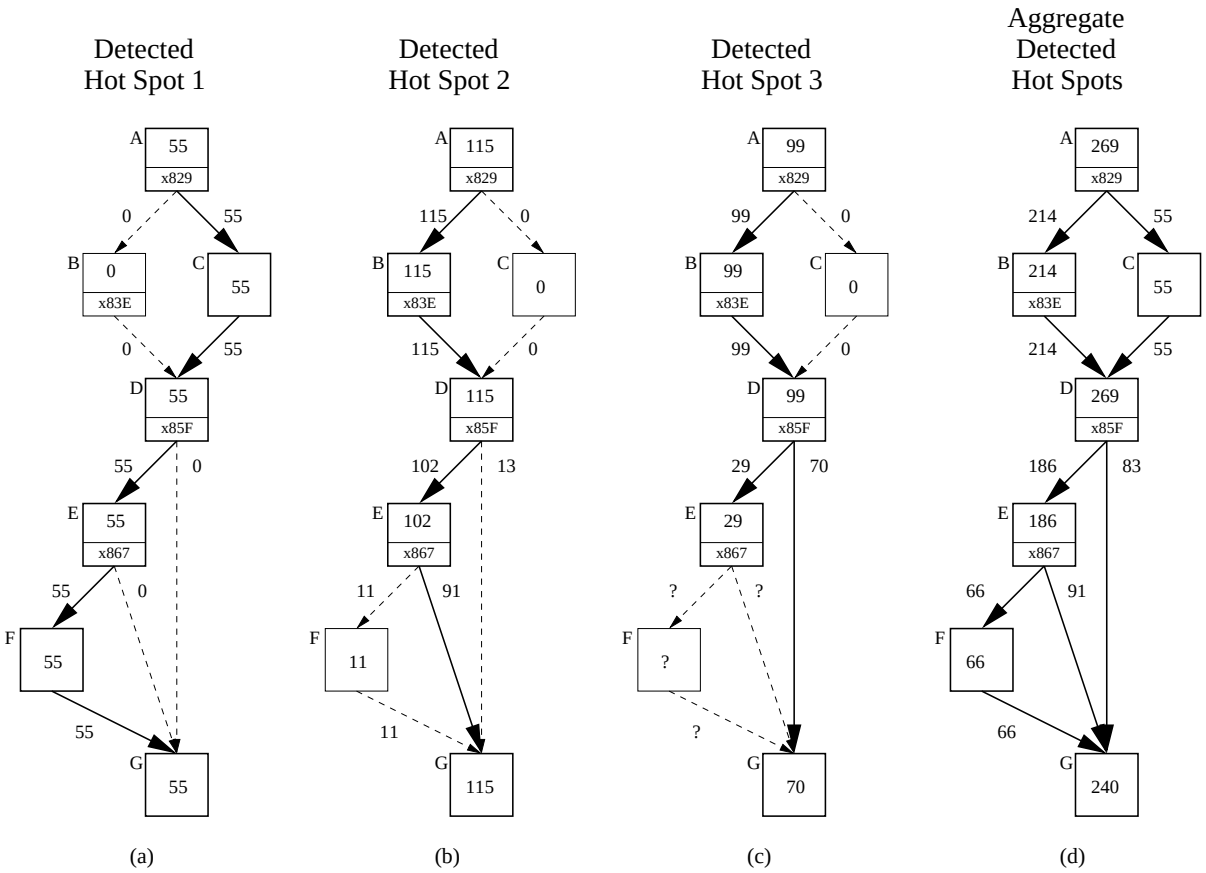


Figure 3.4 Different detected profiles for the `str_new()` function from three different hot spots within *134.perl*.

this function is possible and beneficial. A comparison of the windowed profiles of Figure 3.4 to the overall phase profiles will be discussed in Figure 4.6 of Subsection 4.3.2.

In an effort to better understand the composition of hot spots, another important hot spot was dissected, exposing its control-flow structure. A portion of the primary hot spot from a lisp interpreter (*130.li* from SPEC CPU95) running the training input script is depicted in the control-flow graph in Figure 3.5. This hot spot represents 45% of the program's dynamic execution. Execution flow enters the shown portion of the hot spot at point `A` in function

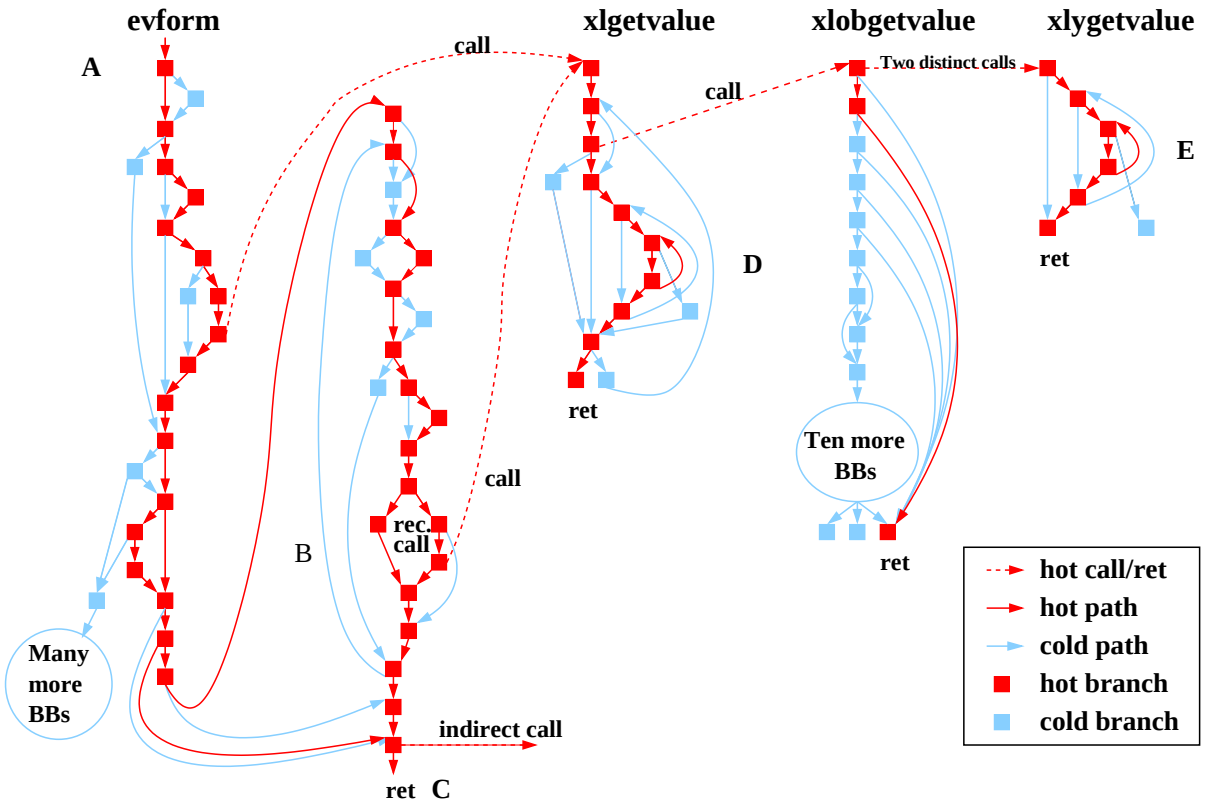


Figure 3.5 Basic block control-flow graph for selected functions in a primary hot spot within *130.li*.

`evform()`, then proceeds through nested functions `xlgetvalue()`, `xlobgetvalue()`, and `xlygetvalue()` before exiting the hot spot at point `C`. Dark (red) boxes and arrows indicate the intensely executed, or hot, blocks and branch paths which are part of the hot spot. The lighter (blue) boxes and arrows indicate the less frequently executed, or cold, blocks and paths which are not. This entire hot spot consists of 81 branches spanning 368 static instructions. The branch profile indicates a primary path through the hot spot, as 47 of the 56 static branches (9.2 million of 10.9 million dynamic branches) have highly consistent dynamic branch direction (greater than 90% in one direction). The hot spot is not simply a tight loop;

rather, control proceeds through a number of functions with minimal inner loops. Notice, for instance, that the loops marked by back-edges `B`, `D`, and `E` iterate only a few times, if at all, during each invocation of the hot spot.

The hot spot behavior witnessed in the lisp interpreter generalizes to other programs as well. While some hot spots do contain tight loops, many also have much more complex control flow. The number of call and return instructions together average about 15% of the dynamic control flow in the examined x86 architecture programs (as described in Section 4.3). While these control transfers are sometimes highly predictable, their frequency presents a barrier to wide instruction fetch and optimization. Many of the benchmarks have a fair number of unconditional jumps, representing, on average, about 5% of dynamic control flow instructions. These instructions perform no control decisions and are obvious candidates for optimization. Indirect jumps and indirect calls do not make up a large portion of the branches, but are frequent enough to become obstacles to long trace formation for some benchmarks. Inlining the potential target may allow for wide fetch across the indirect jump or call, a benefit notably important for calls to shared libraries. Finally, only about 35% of the dynamic control flow instructions fall through to sequential instruction addresses. Consequently, traditional fetch architectures that break fetches at taken branches will often be limited to one basic block per fetch.

Since the proposed architecture is designed to exploit program phases, phase duration and frequency are key issues. ROAR is designed with a minimum phase duration on the order of 50 000 cycles, although this minimum may vary with the number of instructions in the phase. This minimum is required to cover the time required to perform the profiling, extraction, and optimization of the traces, and then to execute the optimized code to reap benefits. What

is termed a program phase in this work could be an inner loop with one set of intensively executed paths, or it could be an outer loop with repeating patterns of shorter inner loops. If the shorter inner loops grow in duration in response to input conditions (for example, the size of the input) they might each become their own phase. The term *phase*, hence, is then associated more closely with a set of paths that execute during a significant fraction of the target temporal window, whether those paths execute uniformly or in spurts.

CHAPTER 4

REGION SELECTION

The first step in the run-time optimization process is region selection. This step is critical because benefits can only be extracted through run-time optimization if the profiled behavior patterns persist. The Hot Spot Detector (HSD) [15] is a hardware profiling structure designed to identify the code regions that comprise the phases of execution, as described in Chapter 3. The detector utilizes a number of new features that provide improvements upon traditional profiling techniques. First, the detector provides profiles on a per phase basis, rather than average profiles collected over the entire execution of the program. As previously described, average profiles can hide behavior patterns specific to various calling contexts and patterns specific to a phase. Second, the detector virtually eliminates overhead associated with profiling. Traditional profiling utilizes instruction sampling that requires frequent interrupts to the operating system to record profile data, or instrumented code that consumes execution resources to record the information. By storing and filtering the profile in a hardware table, no execution overhead is incurred until profile collection is complete.

The Hot Spot Detector provides a number of unique advantages in a run-time optimization system as well. First, the detector provides intense behavior tracking, meaning that every dynamic branch is examined and tracked over a given period. Sampling-based approaches only periodically examine program behavior, often one branch per sample, and may not be

intense enough to accurately profile a phase before execution moves on to another phase. The Hot Spot Detector is designed to track all relevant instructions during execution to clearly delineate behavior. Second, run-time optimization requires low overhead mechanisms. Since the goal of run-time optimization is to reduce execution time, the addition of overhead to perform run-time optimization is counterproductive. Third, examination of the collected profile and determination of a suitable region for optimization is another critical component. This step requires that the mass of profile data be analyzed for suitability. The Hot Spot Detector mechanism unifies the collection and analysis components by providing an integrated on-line filter. Last, the detector provides an approximate relative profile of the behavior within a phase. Whole program behavior analysis is unnecessary because the goal of run-time optimization is to provide performance improvement based on the current execution conditions.

The Hot Spot Detector is designed to be a general mechanism for finding code regions where the most benefit from run-time optimization could be derived. The region selection process is guided by three criteria. First, the region must have a small static code size to facilitate rapid optimization. Second, the instructions in the code region must be active over a minimum time interval so an opportunity exists to benefit from run-time optimization. Third, the instructions in the code region must account for a large majority of the total executed instructions during its active time interval.

4.1 Hot Spot Detector Architecture

The first step in the process of identifying hot spots is to detect the frequently executing blocks of code as they emerge during execution. This process is typically accomplished by monitoring the branches that define the bounds of the hot spot. Employment of a hardware scheme eliminates the time overhead and allows detection to be transparent to the system. The Hot Spot Detector collects the frequently executing blocks by gathering the branches that define their boundaries as well as their relative execution frequency and bias direction. Though not explicitly constructed, a control-flow graph with edge profile weights can be inferred from the collected branch execution and direction information.

Relative to latencies within the processor core, the Hot Spot Detector can tolerate a large latency before recording information about program execution. For this reason, the proposed hardware is off the critical path and gathers required information from the retirement stage. This serves both the purpose of limiting adverse affects on the processor's timing while also preventing the need to handle updates from speculative instructions.

4.1.1 Branch Behavior Buffer

To implement hot spot detection in hardware, we use a cache structure called a *Branch Behavior Buffer* (BBB). The purpose of the BBB is to collect and profile frequently executed branches whose corresponding blocks account for a vast majority of the dynamically executing instructions. Depicted in Figure 4.1, the BBB is indexed on branch address and contains several

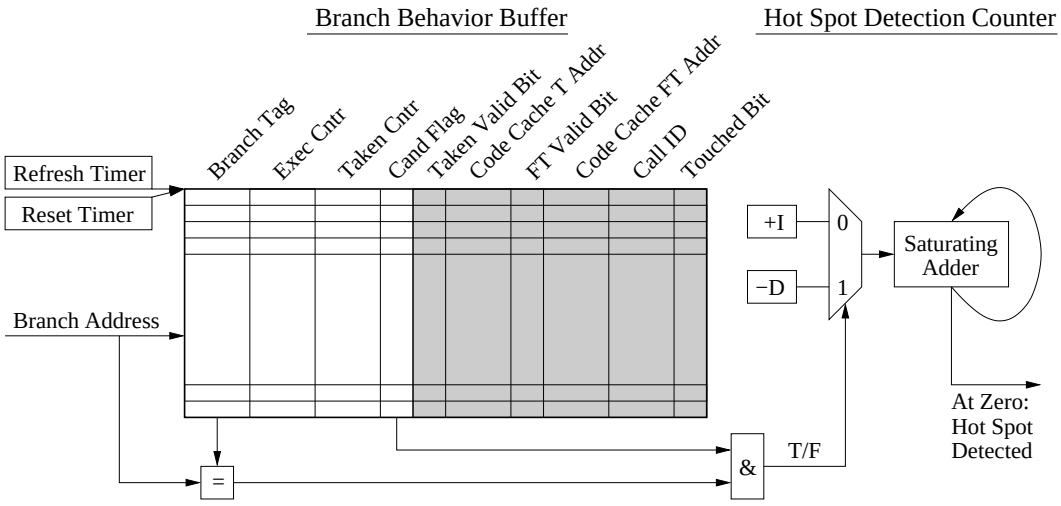


Figure 4.1 Hot Spot Detector hardware. Hot Spot Detector fields shown in white; additional fields for trace layout shown in gray.

fields (detector fields shown in white): *tag* (or branch address), *branch execution count*, *branch taken count*, and *branch candidate flag*.

When the processor retires a branch instruction, the BBB is indexed by the branch's instruction address. If the branch address is found in the BBB, its execution counter is incremented. If the branch is taken, the taken counter is also incremented. To prevent the execution counter from rolling over, the counter saturates at a predetermined maximum value. When the execution counter saturates, the taken counter is no longer incremented in order to preserve the ratio between the taken branch counter and the executed branch counter. As long as the number of branches that reach saturation is small, the profiles will still reflect the relative importance of the branches collected.

A replacement policy must exist to manage entry contention. Because the BBB is a fixed-size table with a fixed indexing scheme, a retired dynamic branch may find neither an existing

entry for its static branch nor a free entry to which it can be mapped. Since the BBB must accurately record the most frequently executed branches and their profiles rather than the most recently accessed, it would be unacceptable to implement a least recently used replacement policy and allow a rare branch to replace a frequently executed branch. Instead, new branches that map to already occupied entries in the BBB are simply discarded. The function of branch replacement is controlled by periodically invalidating some entries.

The entry of branches into the BBB proceeds as follows. When a branch is seen for the first time its behavior is unknown, making it necessary to give the branch a trial period, gather an initial profile, and determine its likely importance. If an entry at its index is available, the branch is temporarily allocated into the BBB and profiled over a short interval called a *refresh interval*. Its execution counter must surpass a threshold called the *candidate threshold* to avoid having its BBB entry invalidated at the next refresh. A branch that surpasses the candidate threshold is called a *candidate branch*, for which the candidate flag in its BBB entry is set, and its entry will not be invalidated at the next refresh.

The refresh interval is implemented using a simple global counter called a *refresh timer* that increments each time a branch instruction is executed. When the refresh timer reaches a preset value, all BBB entries for branches that have not yet surpassed the candidate threshold are invalidated. Refreshing the BBB flushes the insignificant entries and ensures that each branch marked as a candidate accounts for at least a minimum percentage of the total dynamic branches during a fixed interval. The minimum percentage of execution required of candidate

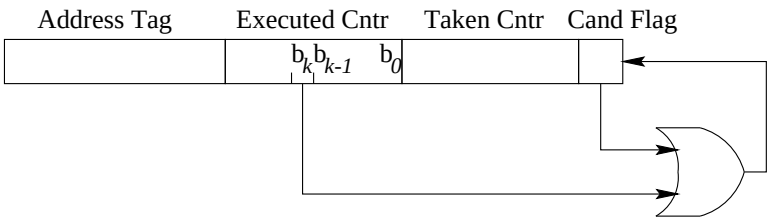


Figure 4.2 Fields in each Branch Behavior Buffer entry.

branches can be expressed as a *candidate ratio*. Thus,

$$CANDIDATE\_RATIO = 2^k/2^n, \tag{4.1}$$

given that the size of the refresh timer is *n* bits, and that the candidate flag is marked when bit b_k of the execution counter is set. An example implementation for a BBB entry with candidate flag is shown in Figure 4.2.

Since entries for low-weight branches are invalidated at each refresh, the BBB only needs to be large enough to hold the candidate branches for a hot spot plus a number of potential candidate branches. If the BBB is too small, the initial allocation of entries to insignificant branches will delay the entrance of important branches into the BBB. Statistically, the important branches will eventually occupy entries after subsequent refresh intervals, but the profile accuracy could be compromised and the reporting of the hot spot branches delayed.

A more serious conflict occurs when two important branches index into the same cache location. Many of these conflicts can be eliminated by making the BBB set-associative. However, some conflicts will still exist. As long as the remaining conflicts are relatively rare, a run-time optimizer can be designed to infer the presence of the missing branches. As discussed

in Chapter 5, ROAR utilizes actual traces of execution to fill these information gaps. Kirchoff's current law, which states that the flow into any node (block) must equal flow out of the node, could also be used to derive the profile weights of a single missing branch given that the block and branch weights of the surrounding code are accurate.

4.1.2 Hot Spot Detection Counter

Once candidate branches have been identified in the Branch Behavior Buffer, they must be monitored to determine whether the corresponding blocks may be considered a hot spot region and, thus, useful for optimization. We define a threshold on the minimum percentage of candidate branches that must execute over a time interval as the *threshold execution percentage*, and we define the actual percentage of candidate branches executed over an interval as the *candidate execution percentage*. Two criteria must be satisfied before a group of candidate branches are declared to accurately define the bounds of a hot spot. First, the candidate execution percentage should equal or surpass the threshold execution percentage. Second, this high candidate execution percentage should be maintained for some minimum amount of time. When the two criteria are met, a detection occurs, and all table entries are frozen.

In order to minimize disruption of the system during the hot spot detection process, we track the behavior of the candidate branches in hardware using a *Hot Spot Detection Counter* (HDC), shown in Figure 4.1. The Hot Spot Detection Counter is implemented as a saturating up/down counter that is initialized to its maximum value. It counts down by D for each candidate branch executed or counts up by I for each noncandidate branch executed, where the values of D and I along with the counter size are determined by the desired thresholds, as will be discussed in

Subsection 4.1.3. When the candidate execution percentage exceeds the threshold execution percentage, the counter begins to move down because the total amount decremented exceeds the amount incremented. Essentially, the candidates are engaged in a tug-of-war with the noncandidates where the value of the HDC indicates which side is winning. If the candidate execution percentage remains higher than the threshold for a long enough period of time, the counter will decrement to zero. At this point, the BBB entries contain the skeleton of the hot spot which will be preserved until an optimizer is called to read the table, construct the region, and perform the optimization. This process may include a software run-time optimizer that would be invoked via the operating system, or a hardware run-time optimizer (such as the Trace Generation Unit in the next chapter) enabled directly by the detection mechanism.

The difference between the candidate execution percentage and the threshold execution percentage determines the rate at which the counter decrements (i.e., the rate at which the hardware identifies the hot spot). This corresponds to the observation that hot spots become more desirable as they either account for a larger percentage of total execution or run for a longer period of time. It is assumed that hot spots that have been active over a longer period of time are less likely to be spurious in their execution and are more likely to continue to run after optimization has been completed.

There are three primary scenarios where there is no hot spot to be found, and thus the HDC will never reach zero:

1. Few branches execute with sufficient frequency to be marked as candidates, and collectively, they do not constitute a large percentage of the total execution. Thus, even if the region they bound were classified as a hot spot and optimized, only a small benefit is

likely to materialize since so few cycles would be spent inside the region. This situation could be referred to as *wandering* execution where large portions of the application are touched and not revisited in any particular pattern.

2. The number of branches that execute frequently enough to be considered candidates is too large to fit into the BBB. If the branches that are able to enter the BBB do not account for a large enough percentage of execution, they will not indicate the presence of a hot spot. This may happen if the execution profile of the region is very flat, and may appear to be wandering when there is actually more structure to the execution pattern. Branches in large loops would tend to have more biased and repeatable behavior, unlike wandering code. Although some benefit may be gained by optimizing all the frequent branches, the overhead of optimizing such a large region could be prohibitive. However, the presence of a large loop may indicate lengthy execution within the loop and thus might justify a strategy that would focus BBB profiling on a portion of the loop. Generally, loops that are larger than the instruction cache also result in poor front-end processor performance, and further study on effective means for handling them is warranted.

3. The execution profile is not consistent. In this case, a small set of branches may account for a large percentage of execution over a short time, but execution shifts to a different region of code before the Hot Spot Detection Counter saturates. Optimizing a region of code that only executes spuriously is unlikely to yield much benefit.

In each of these scenarios, some branches were executed frequently enough to warrant candidate status and therefore consideration for inclusion in a hot spot and for further profiling.

However, if the collection of candidate branches does not identify a hot spot region after continued tracking, all branches must be cleared in order to begin a fresh detection process. In other words, some branches may have once been important, but now are cluttering the detection attempt. Therefore, the BBB will be periodically purged by the *reset timer* to make room for new branches. This timer is similar to the refresh timer but clears all entries in the BBB, including candidate branches. The reset interval should be large enough to allow the HDC to saturate for valid hot spots but small enough to allow quick identification of a new phase of execution. During a reset, a number of threshold values and parameters in the detector could be dynamically updated to alter the hot spot search criteria.

It should be noted that the weights of the blocks and edges are only valid within a particular hot spot. Although this profile information is useful for inferring a control-flow graph for a particular hot spot, profiles of different hot spots cannot be meaningfully compared. For instance, even if one hot spot has block weights twice those of another hot spot, the first hot spot is not necessarily executed twice as often or for twice as long as the second. These weights are primarily affected by two factors: refresh periods required to detect the hot spot, and frequency of the particular instructions within the hot spot. The greater the number of refreshes required to detect a hot spot, the longer the profiles are allowed to accumulate, and the greater the weights will be.

4.1.3 Hot spot detection parameters

Once the threshold execution percentage X_t required for hot spot detection has been selected, the HDC increment and decrement values should be chosen. *D* is the decrement value

for a candidate branch (*candidate hit*), and *I* is the increment value for a branch that is not in the table or is not yet marked as a candidate (*candidate miss*). Let *X* be the actual candidate execution percentage. For a given *D* and *I*, the counter will decrease when the candidate execution percentage multiplied by the decrement value is greater than the percentage of non-candidates multiplied by the increment value. This is represented by the equation

$$X * (-D) + (1 - X) * (I) \leq 0 \tag{4.2}$$

Rearranging the terms and solving for *X* yields the formula for minimum percentage:

$$X \geq \frac{I}{D + I} \equiv X_t \tag{4.3}$$

Equation (4.3) shows that the counter decreases when the percentage of execution is above the threshold, as determined by *I* and *D*.

Given the increment and decrement values, the size of the HDC can be chosen to achieve a minimum detection latency. Let *N* be the minimum number of branches executed before a hot spot is detected. For detection to occur, the following inequality must hold:

$$N * X * (-D) + N * (1 - X) * (I) \leq -HDC\_MAX\_VAL \tag{4.4}$$

Thus, the latency for detecting a hot spot is determined by the following equation:

$$N = \frac{HDC\_MAX\_VAL}{(D + I) * (X - X_t)} \tag{4.5}$$

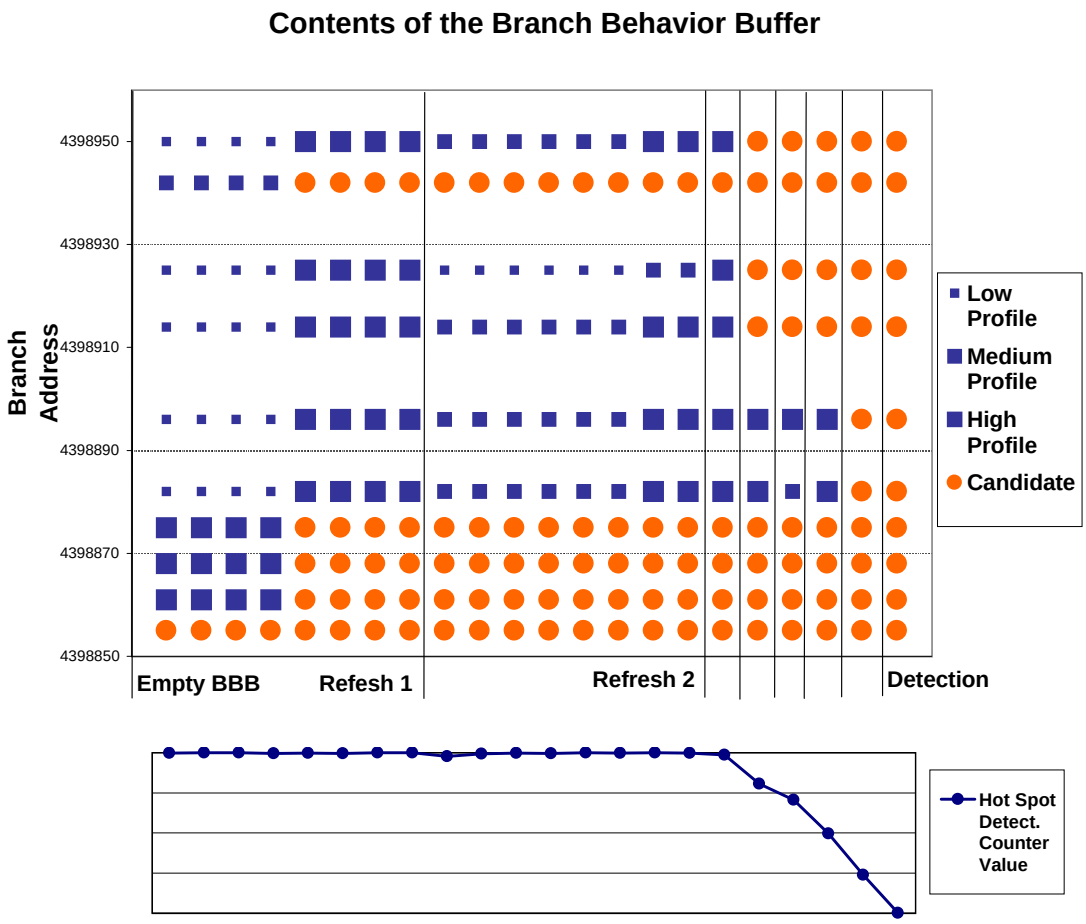


Figure 4.3 Example of the hot spot detection process.

As the candidate execution percentage further surpasses the threshold, the detection latency decreases. The latency can also be decreased independently of the candidate execution percentage by increasing *I* and *D* such that X_t remains constant.

4.1.4 Hot spot detection example

A graphical example of the hot spot detection process taken from hot spot 4 in *134.perl* (primary hot spot 2 in Figure 3.1) is shown in Figure 4.3 (comprehensive detection results will

be presented in Section 4.3). In the figure, refresh time intervals are shown across the top graph with the thin, black vertical lines indicating the end of each refresh interval when noncandidate branches are flushed from the table. Eight data samples were taken at evenly spaced intervals for the first two refresh periods to highlight the process, while a single data sample was taken just prior to the refresh for the other intervals. The top graph charts the state of the BBB executed counters for a subset of the active instruction addresses in the hot spot. A dark (blue) square marker indicates that the branch at the given address had not yet been classified as a candidate. However, the larger the square, the closer it was to becoming a candidate. Note that at the end of the refresh intervals, noncandidate branches (marked with squares) are cleared from the table. Had these branches been spuriously executed in this phase, they would have been cleared from the BBB after the refresh, allowing other branches that index to the same location an opportunity to be considered for inclusion in the hot spot. If the branches are re-encountered and BBB entries are available, profiling will be reinitiated, but back at weights of zero. This situation can be observed for the top branch `4398950` which continued execution, albeit infrequently, in the second interval and is therefore represented by a reduction in square size at the beginning of interval 2. Candidate branches are shown in the graph as light (orange) circles, which are locked into the table across refresh intervals. In this example, portions of the hot spot (mainly the branches at the bottom of the graph) were heavily executed throughout the hot spot and were quickly classified as candidates. However, a number of the other branches, notably the two centered around address `4398890`, required more time to prove themselves important.

The bottom graph shows the progress of the Hot Spot Detection Counter over the same refresh intervals. During the first three intervals, a number of the important branches executed sporadically and had not been identified as candidates. As previously described, these noncandidates caused increments in the HDC. Clearly, the noncandidates were dominant in the first intervals because the HDC in the graph remains saturated at the highest value. However, as the noncandidates achieved candidate status and began to cause decrements to the counter (interval four), the counter began its descent to zero and eventual hot spot detection.

4.1.5 Monitor Table

Coupled with the HSD is a global table called the *Monitor Table* [1]. The purpose of the Monitor Table is to determine when hot spot profiling is necessary. It is only used in the experiments in this chapter as other mechanisms are used in the complete ROAR system. This hardware mechanism is continuously running, watching program execution and comparing the executing branches to those already determined to be in hot spots. When the program is executing in the known hot spots, the system is said to be in *monitor mode*, which is the steady-state mode of execution. The system enters *profile mode* and the BBB is enabled when the Monitor Table determines that execution has strayed from the known set of hot spots. Note that the Monitor Table continues to operate during profile mode, watching for execution to return to the set of known hot spots. If this situation were to occur, the BBB would be deactivated, since it is unnecessary and costly to extract and possibly optimize a hot spot that has already been processed.

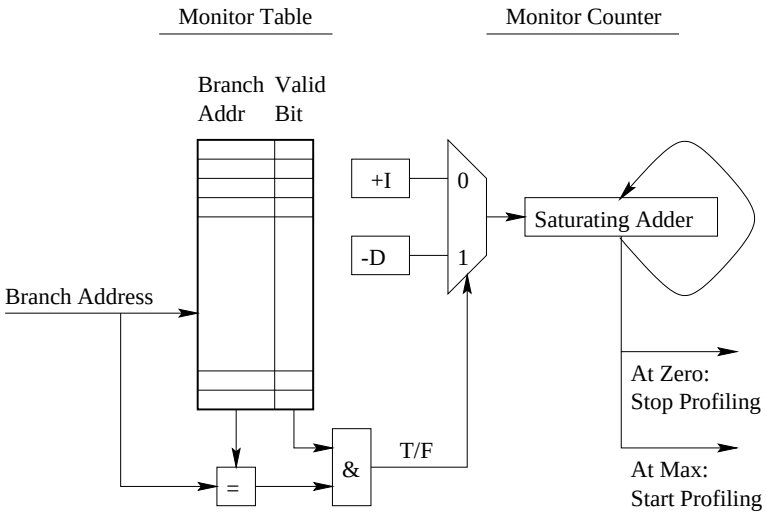


Figure 4.4 Monitor Table hardware.

In order for the system to detect when execution strays from the known set of hot spots, the hardware must be aware of those hot spots which have already been identified. In an ideal Monitor Table, the addresses of all branches in all known hot spots would be placed into a tag array as shown in Figure 4.4. When a branch is executed, the instruction address is used to index in the tag array. Its presence in the array indicates execution in a hot spot. An up/down counter called the *Monitor Counter* is used to track long-term execution trends and operates much like the HDC. It counts down when a hot spot branch is executed and counts up when a non-hot spot branch is executed. When the Monitor Counter saturates at the maximum value in monitor mode, a high percentage of recent branches outside of the known hot spots have been executed, indicating a possible transition to a new hot spot. At this time, profile mode is resumed. Similarly, when the system is in profile mode and the Monitor Counter reaches zero, program execution must have returned to the set of known hot spots. At this time, the BBB is deactivated, and monitor mode is resumed. When monitor mode is entered from profile mode

(either the Monitor Counter just saturated at zero to end profile mode, or a hot spot was just detected and execution is likely to continue in the hot spot) the Monitor Counter is initialized to zero, indicating that execution is in a hot spot.

As in the HDC, the increment and decrement values for the Monitor Counter determine the threshold ratio of hot spot to non-hot spot branches. Although a minimum ratio of hot spot to non-hot spot branches must be maintained to remain in monitor mode, this ratio should not be as high as in the HDC. A lower ratio is used for the monitor hardware to allow the behavior of the hot spots to vary slightly without reentering profile mode. Once this ratio is determined, the same formula used for the HDC can be used to derive suitable increment and decrement values for the Monitor Counter.

4.1.6 Multiprocess support

Thus far, the hardware design has assumed single process execution. Operation becomes slightly more complicated when considering the context switching abilities of microprocessors. It is the responsibility of the operating system to correctly maintain the state of the proposed hardware in a multiprocess environment as it is required to do for traditional hardware components.

Because of the expense of swapping out even a subset of the BBB during context switches, the hot spot detection hardware is designed to operate in single process mode. The BBB is a processor resources that is allocated to a particular process. It maintains the branches for the designated process until a detection or a clear, even across context switches. Because the BBB responds to hot spots quickly, it is only active during a small percentage of an application's

execution. Because of this low utilization, a single BBB can be shared among multiple processes. The BBB could be configured to search for hot spots associated with a particular process ID, thread ID, or code segment. This ID would be stored in a control register. Alternatively, if the context switches were infrequent, BBB profiling could begin anew for each process immediately after the process is swapped into the processor.

Unlike the BBB, the Monitor Table is always in use by each process. Since swapping a table in and out at each context switch would be extremely costly, all processes could share a single table. In order to accomplish this, an up/down Monitor Counter is necessary for each process or active subset to effectively track hot spot behavior. Again, if context switches were infrequent, a single Monitor Counter could be initialized to zero immediately after each context switch. Furthermore, each entry in the Monitor Table must also be tagged with its process ID. This ID serves as a tag for comparison purposes when determining a hit or miss and for determining which Monitor Counter to update.

When a Monitor Counter saturates indicating that profiling is necessary, the BBB must first be allocated to that particular process. A simple check of the BBB process ID control register can be made to determine BBB ownership. If the requesting process owns the BBB, profiling can continue without delay. Otherwise, arbitration must occur between all of the processes requesting use of the BBB. This arbitration may be implemented in the hardware itself or within the OS. Although processes may be denied use of the BBB for a short time, the BBB may be acquired by the next waiting process as soon as the BBB resets or the HDC saturates.

4.1.7 Enhancements to the base hardware

Several enhancements might be made to the base hardware. The first enhancement would be to reduce the size of the BBB by considering only conditional and indirect branches. The execution and taken profile weights of blocks with unconditional branches or direct calls can be determined by inference via Kirchhoff's First Law and need not be profiled. However, this approach may require a runtime optimizer to spend more time analyzing and constructing the control-flow graph.

A second enhancement would be to index into the BBB with a combination of the branch's address and its direction. Thus, the taken and not taken paths of a particular branch would be recorded in separated entries allowing for the elimination of the taken counter in each BBB entry. While this approach would result in either an increase in the size of the BBB or a reduction in the number of distinct branch addresses in the BBB, it would allow the system to detect finer changes in program, and hot spot, behavior. For example, control flow changes within a particular set of blocks would now be detected and possibly reoptimized.

A third enhancement would be to include support for profiling the arc weights of indirect branches. Currently, profiling determines only the branch weights of these branches. An additional table indexed on a combination of the branch and target addresses could be used to store the actual profile. The BBB entry for that branch would still gather the branch execution count and determine branch candidacy.

Table 4.1 Hardware parameter settings.

Parameter	Setting
Number of Branch Behavior Buffer sets	1024
Branch Behavior Buffer associativity	2-way
Executed and taken counter size	9 bits
Candidate branch execution threshold	16
Refresh timer interval	4096 branches
Clear timer interval	65 535 branches
Hot Spot Detector counter size	13 bits
Hot Spot Detector counter increment	2
Hot Spot Detector counter decrement	1
Monitor Table counter size	12 bits
Monitor Table counter increment	1
Monitor Table counter decrement	1

4.2 Hot Spot Detection Experimental Setup

Because the design space is large, experimentally evaluating the individual effect of each hardware parameter was infeasible. Initial parameters were selected that attempted to match the observed hot spot behavior and were then further refined, resulting in parameters that exhibit desirable hot spot collection behavior. These parameters were used in the experiments presented in this section and are shown in Table 4.1. The BBB hardware was configured to allow branches with a dynamic execution percentage of 0.4% (16 executions/4096 branches) or higher to become candidates (the candidate ratio). The HDC was configured with a threshold execution percentage that required candidate branches to total more than 66% (2:1) of the execution to indicate a hot spot. The BBB was allowed 16 refreshes (totaling 65 535 branches) to detect a hot spot before it was reset. These parameters were chosen through empirical study comparing the quick detection needs of smaller hot spots to the profiling breadth requirements

of larger ones. The threshold execution percentage, as previously described, is the starting point for parameter determination and is set to allow for some spurious execution during the detection process.

4.3 Hot Spot Detector Evaluation

Trace-driven simulations were performed on a number of applications in order to explore the effectiveness of the Hot Spot Detector. The experiments for the detector were designed to maximize the program coverage of the collected hot spots while minimizing both the number of spurious branches in the hot spots and the latency of detection. Both SPEC CPU95 and common WindowsNT applications were simulated to provide a broad spectrum of typical programs. These benchmarks are summarized in Table 4.2. The eight applications from the SPECINT95 benchmark suite were compiled from source code using the Microsoft VC++ 6.0 compiler with the *optimize for speed* and *inline where suitable* settings. Several WindowsNT applications executing a variety of tasks were also simulated. These applications are the general distribution versions, and thus were compiled by their respective independent software vendors.

The experiments were performed using the inputs shown in Table 4.2. In order to extract complete execution traces of these applications (all user code, including statically and dynamically linked libraries), we used SpeedTracer, special hardware capable of capturing dynamic instruction traces on an AMD K6 platform [16]. Since the traced instructions are from the x86 instruction set architecture (ISA), variable-length instructions are used throughout simulation.

Table 4.2 Benchmarks for detection and trace generation experiments.

Benchmark	Num. Insts.	Actions Traced
099.go	89.5M	2stone9.in training input
124.m88ksim	120M	clt.in training input
126.gcc	1.18B	amptjp.i training input
129.compress	2.88B	test.in training input *count* enlarged to 800k
130.li	151M	train.lsp training input (6 queens)
132.ijpeg	1.56B	vigo.ppm training input
134.perl	2.34B	jumble.pl training input
147.vortex	2.19B	vortex.in training input
MSWord(A)	325M	open 16.0 MB .doc file, search, then close
MSWord(B)	911M	load 25 page .doc, repaginate, word count, select entire doc, change font, undo, close
MSExcel	168M	VB script generates Si diffusion graphs
Adobe PhotoDeluxe(A)	390M	load detailed tiff image, brighten, increase contrast, and save
Adobe PhotoDeluxe(B)	108M	exported detailed tiff image to encapsulated postscript
Ghostview	1.00B	load gsview and 9 page ps file, view, zoom, and perform text extraction

To ensure examination of all executed user instructions, sampling was not used during trace acquisition or simulation.

In order to evaluate the performance of the Hot Spot Detector, a number of experiments were conducted to examine the dynamic and static instruction coverage of the hot spots produced. Since the detected hot spots provide the basis for trace generation and optimization, maximizing coverage of the dynamic execution with a small number of hot spots is critical. While the trace generation process utilizes heuristics to account for an occasional missing important branch, neglecting branches will often preclude optimization of paths containing those

Table 4.3 Summary of the hot spots found in the benchmarks.

Benchmark	# hot spots	# static insts. in hot spots	% static executed insts. in hot spots	% total exec. in hot spots	% total exec. in detected hot spots	Dyn. insts. in hot spots after detection
099.go	6	2398	3.46	37.84	35.39	31.7M
124.m88ksim	4	1576	2.78	93.03	92.30	110M
126.gcc	47	17665	8.90	58.42	52.12	617M
129.compress	7	918	2.12	99.93	99.81	2.87B
130.li	8	1447	3.00	91.28	90.88	137M
132.ijpeg	8	2556	3.48	91.07	91.00	1.42B
134.perl	5	1738	2.13	88.43	85.99	2.01B
147.vortex	5	2161	1.76	72.30	71.93	1.58B
MSWord(A)	5	3151	1.17	91.36	91.08	296M
MSWord(B)	21	12541	2.40	69.13	62.04	566M
MSExcel	25	18936	2.94	60.01	54.85	88.2M
PhotoDeluxe(A)	20	5485	1.68	94.31	90.97	354M
PhotoDeluxe(B)	14	4192	1.78	94.24	90.81	98.5M
Ghostview	33	8938	2.82	73.39	72.55	2.30B

branches. However, providing concise hot spots as swiftly as possible ensures minimal optimization overhead and maximum available time to spend in optimized code.

4.3.1 Hot spot detection coverage

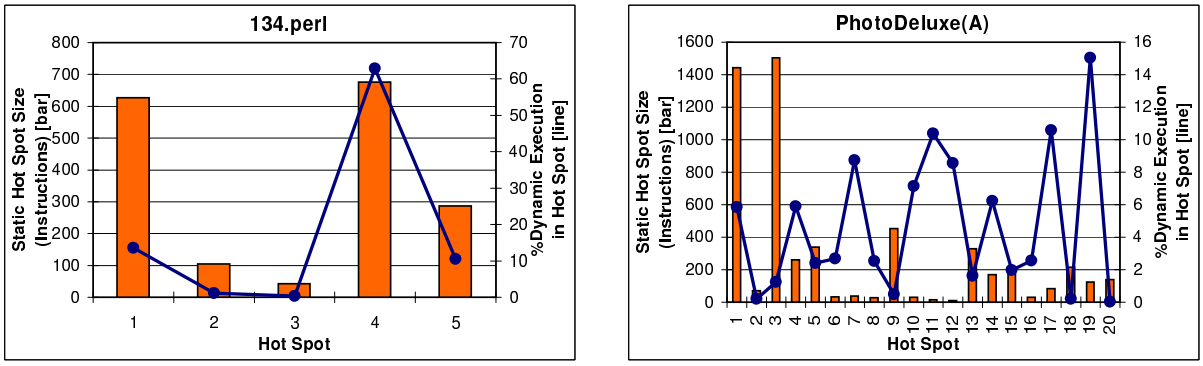
Table 4.3 summarizes the effectiveness of the proposed hardware at detecting run-time optimization opportunities for each benchmark. The *number of hot spots* column lists the number of times that the Hot Spot Detection Counter saturated at zero, indicating the detection of a new hot spot. The detector was reactivated each time execution drifted from the cumulative set of candidate branches from the previously detected hot spots. The *number of static instructions in hot spots* is the total number of instructions that will be delivered to the optimizer collectively

over the entire execution of the program. The next column, *percent static executed instructions in hot spots*, relates the number of static instructions in hot spots to the total number of static instructions executed. The results show that, out of all the static instructions executed by the microprocessor, only a small percentage lie within hot spots. Note that some instructions may be present in more than one hot spot and, accordingly, are counted multiple times. The portion of total dynamic instructions represented by these hot spots is shown in the next column, *percent total execution in hot spots*. Because this hardware cannot detect hot spots instantly, some time that could be spent executing in optimized hot spots is spent executing original code during detection. The time spent in hot spots after they are detected is shown in the *percent total execution in detected hot spots*, and the time lost to detection can be found by taking the difference between this column and the previous column. Finally, the last column, *dynamic instructions in hot spots after detection*, shows the number of dynamic instructions that could benefit from run-time optimization. This number reflects any subsequent reuses of detected hot spots.

Analysis of the results shows that only a small percentage, usually less than 3%, of the static code seen by the microprocessor executes intensively enough to become hot spots. Since a large percentage of the dynamic execution is represented by a small set of instructions, often nearly 90% of the program's execution, a run-time optimizer can easily focus on this small set with the potential for significant performance increase. In addition, only about 1% of the possible time spent in optimized hot spots is missed due to the detection process. There is no actual overhead since the hardware performs the profiling in parallel with unhindered execution. For example, in *130.li*, the number of hot spot static instructions comprises only

3% of the total static instructions, yielding a total hot spot code size of 1447 instructions. Furthermore, 90.88% of the entire execution is spent in detected hot spots. Analysis shows that ideally the hot spots account for 91.28% of execution, and thus only 0.40% is lost during the detection process. This indicates that the Hot Spot Detector makes the identification so swiftly that the execution of hot spot regions falls almost entirely within potentially run-time optimized code.

Examining individual hot spots reveals interesting characteristics of program behavior. Figure 4.5(a) details the detected hot spots from the *134.perl* benchmark. For each hot spot, the bar graph shows the static code size of the hot spot, while the line graph shows the percentage of execution spent in that hot spot after detection. This benchmark consists of three *primary* hot spots (loosely defined as those that represent at least 1% of total application execution): 1, 4, and 5 on the graph. These correspond to the three hot spots of Figure 3.1 in Chapter 3 (note that in the histogram, *134.perl* was compiled for the IMPACT [17] architecture without inlining). From Figure 3.1, code can be seen executing between the first and second primary hot spots, which correspond to hot spots 2 and 3 in Figure 4.5(a). In the third primary hot spot, hot spot 5 of Figure 4.5(a), the `cmd_exec()` function loops 117K times, calling `str_free()` in each iteration. The 9 blocks, totaling 43 static instructions, contribute 7.3M dynamic instructions to the program's total execution. Because these blocks execute intensely, the hot spot serves as a good candidate for run-time optimization. Analysis of this benchmark also shows that one hot spot is much more dominant than the others in terms of dynamic execution. In this case, optimizing only hot spot 4 could benefit over 58% of the dynamic instructions executed.



(a) 134.perl. (b) PhotoDeluxe(A).

Figure 4.5 Detailed hot spot statistics.

Similar characteristics were observed in the other benchmarks. Figure 4.5(b) shows an example from one of the precompiled WindowsNT applications, *PhotoDeluxe(A)*. For this benchmark, there are several hot spots that each represent at least 8% of the total execution and together represent more than 50%. We also see quite a few hot spots with small static code sizes, indicating tight, intensely executed code. In fact, for this benchmark, the smaller-sized hot spots are also those with high total execution percentages, indicating excellent opportunities for run-time optimization.

The benchmark *099.go* is a notable example of a benchmark without obvious hot spots. While this game simulation repetitively executes players' moves, each move touches a large amount of static code with little temporal repetition. The hardware was still able to detect six hot spots representing 35% of the execution. There is one primary hot spot that represents 28% of the execution with a static code size of 1170 instructions. Data has shown that the static sizes of the detected hot spots vary significantly, from tens of instructions to the low

thousands. Detailed hot spot detection results for the benchmarks in Table 4.3 are presented in Figures B.1-B.8 in Appendix B.

4.3.2 Hot spot detection accuracy

To evaluate the accuracy of the hot spot profiler, the branch taken ratios collected at hot spot detection-time were compared to their average taken ratios at the completion of the application. The completion-time statistics were gathered by attributing each dynamic branch execution to the appropriate hot spot.

As previously mentioned, the weights of the blocks and edges are comparable within a particular hot spot, allowing for the construction of a meaningful control-flow graph. However, these weights cannot be meaningfully compared between hot spots. Referring back to Figure 3.4 in Section 3.2, the Hot Spot Detector found three primary hot spots. The profile weights gathered by the detector are annotated on the blocks and arcs. Even though the block weights in Hot Spot 3 are twice as large as the weights in Hot Spot 1, it does not mean that Hot Spot 3 is executed twice as often as 1. These weights are primarily affected by two factors: refresh periods required to detect the hot spot, and frequency of the particular instructions within the hot spot. The greater the number of refreshes required to detect a hot spot, the longer the profiles are allowed to accumulate, and the greater the weight values will be. Furthermore, if the instructions are part of a tight loop, the profile weights will accumulate to a greater value during the detection time than will instructions that are executed as part of a larger region of code.

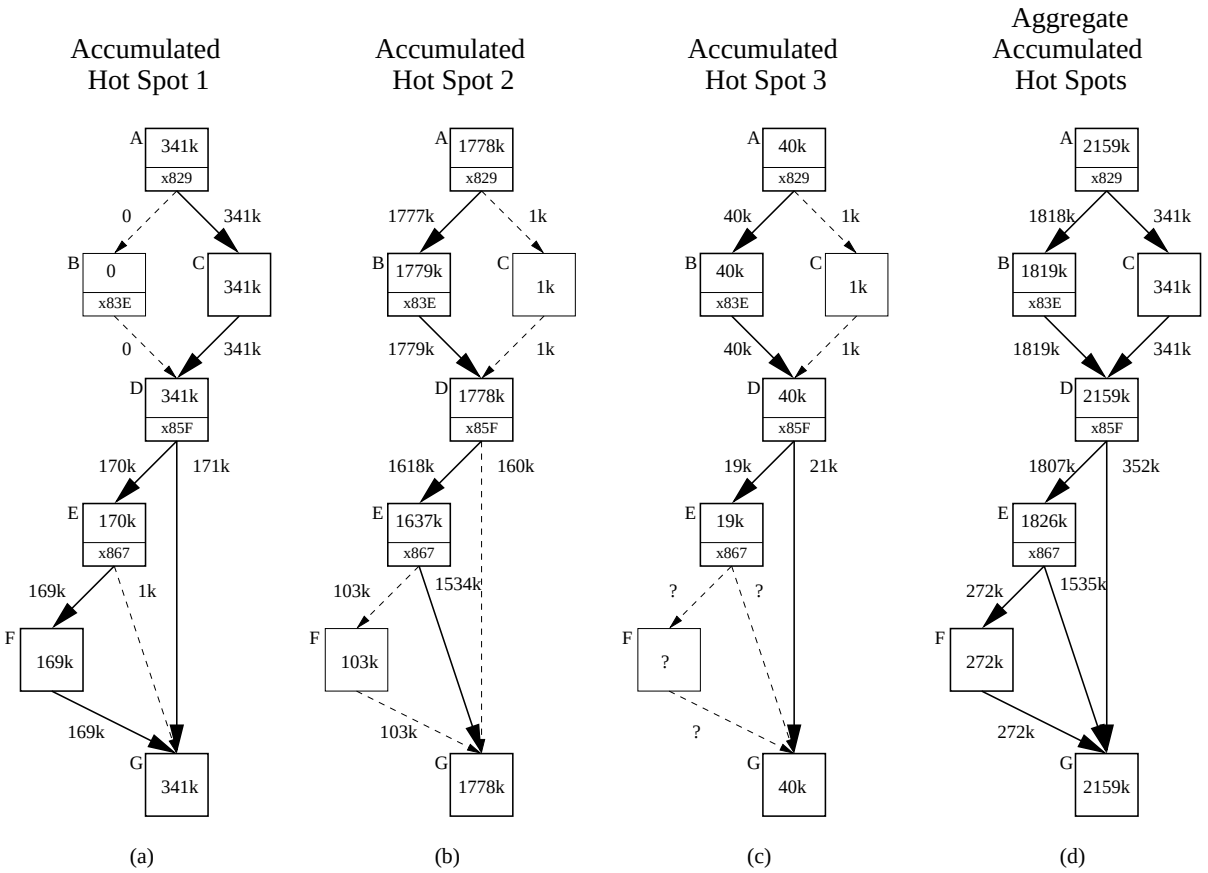


Figure 4.6 Different accumulated profiles for the `str_new()` function from three different hot spots within *134.perl*.

Figure 4.6(a)-(c) depicts the actual weights of the profiled blocks and edges. These weights are gathered through a heuristic that attributes instruction executions to the hot spots. The heuristic assigns a dynamic instruction execution to a hot spot if that static instruction was a candidate at detection time. When a static instruction is part of multiple hot spots, it is attributed to the hot spot that was uniquely identified by the previous dynamic branches. While the detected weights of Hot Spots 2 and 3 are almost the same, the accumulated weights are heavily skewed toward Hot Spot 2. However, both phases last considerably longer than the

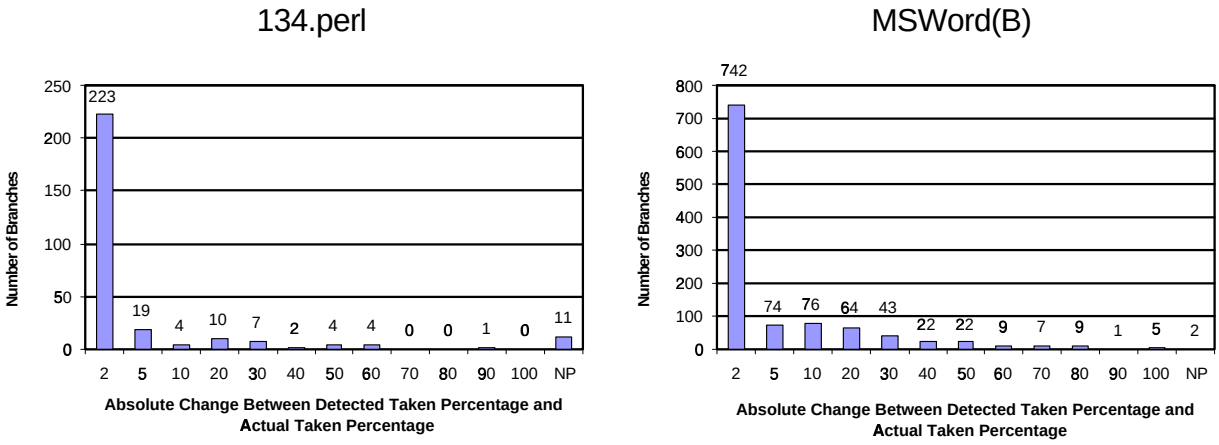


Figure 4.7 Absolute difference between detected taken percentage and actual taken percentage for the hot spot branches in all significant detected hot spots.

profiling process. For block E in Hot Spot 3, the profile weight is missing from the hot spot, as previously mentioned. Since the heuristic must attribute the execution of those blocks to some hot spot, a corresponding increase of 19 000 executions can be seen in the same block in Hot Spot 2 as compared to the weight of its incoming edge.

Figure 4.7 shows the comparison between the taken ratios for the hot spot branches in two typical benchmarks. In these bar charts, only hot spots that represented 1% or more of the program execution were considered. For each hot spot branch, the difference between its detected taken percentage and its accumulated counterpart is computed and tallied. The figure shows that a vast majority of the branches have taken ratios at detection time within 2% of their accumulated values. These results indicate accurate profiling and therefore the potential for using detection profile weights for optimization. However, a few branches also change their behavior dramatically as can be seen by the larger changes. The last category of the graph, not present (NP), represents branches that were detected as part of the hot spot

but never executed after detection, essentially false positives. Including these branches when performing optimization may introduce obstacles to aggressive optimization. These obstacles include wasted optimization on code that will never be executed, false paths that may partially overlap with real paths thus preventing the real paths from being formed, or new side entrances into existing traces that may force existing traces to be split.

4.3.3 Hot spot detection quality

The size of the Branch Behavior Buffer is an important factor because it represents the maximum size of any hot spot in terms of branches. Even if the capacity of the BBB is adequate for the final hot spot, the buffer must also contain enough entries for spurious branches as well, so that the spurious ones do not cause significant contention with the important branches. Table 4.4 shows the detection information for the *MSWord(A)* application. Hot spot detection was performed on the application four times with settings of 2048, 1024, 512, and 256 entries in the BBB, each with four way set associativity. The detected hot spots that represent over 1% of the application execution are listed A - C.

For the 2048 entry BBB, three primary hot spots were detected as opposed to two for the other configurations. The third hot spot in this configuration is the result of a slight shift in HSD timing that results in the detection of a hot spot that contains portions of the first two. Figure 4.8 depicts the Venn diagrams for the 2048 and 1024 entry configurations. The two hot spots in the 1024 entry configuration are largely disjoint, whereas the third hot spot in the 2048 configuration is comprised mostly of pieces of the first two. However, the shared branches between the first and third hot spots are the most heavily executed, thus shifting execution

Table 4.4 Detection summary for MSWord(A).

Hot Spot	BBB Entries	2048	1024	512	256
A	Number of Operations in Hot Spot	359	359	359	339
	BBB Branches Seen	20 280	20 280	21 252	22 151
	Dynamic Conflicts	18	94	995	2810
	Set Conflicts	5	31	84	64
	Percent of Execution	3.39	7.79	7.79	7.68
B	Number of Operations in Hot Spot	1012	1009	972	892
	BBB Branches Seen	14 456	14 651	13 451	22 455
	Dynamic Conflicts	96	881	418	5083
	Set Conflicts	8	43	13	41
	Percent of Execution	75.92	83.16	81.92	70.70
C	Number of Operations in Hot Spot	554			
	BBB Branches Seen	12 129			
	Dynamic Conflicts	0			
	Set Conflicts	0			
	Percent of Execution	11.84			
TOTAL	Percent of Execution	91.15	90.95	89.71	78.38

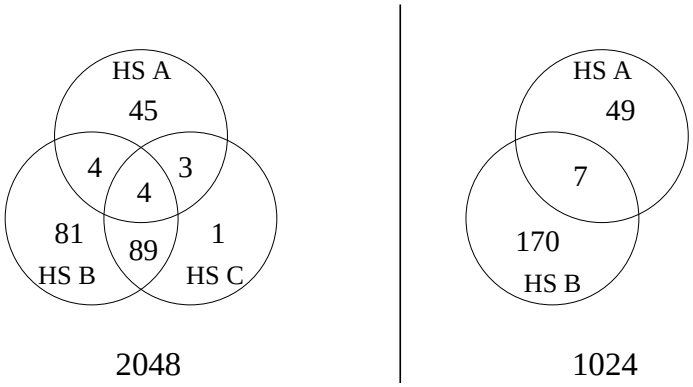


Figure 4.8 Venn diagrams depicting the number of shared branches in the hot spots for the 2048 and 1024 BBB entry configurations.

percentage from the first to the third, as compared to the 1024 configuration. This hot spot represents the code pattern around the transition between the first two.

The total application coverage (bottom row of Table 4.4) drops slightly from above 91% to just above 78%. For the first hot spot, the number of operations in the hot spot is about the same for each configuration, but drops by about 5%, for 256 entries, although the percentage of execution drops by only 1.5%. The number of dynamic branches seen during the detection process also climbs as the buffer becomes smaller, from approximately two refreshes to close to three. In this experiment, the reset interval is set to eight refreshes, which is still significantly larger than needed. The number of dynamic branches that are dropped due to capacity limitations and the number of sets affected by capacity limitations are also higher with the smaller BBBs. Essentially, a 256-entry buffer has the capacity to detect a hot spot that is about 340 branches in size, although it takes a bit longer due to contention than with a larger buffer. For the second hot spot, similar behavior is observed, except that the application coverage is more significantly decreased when the buffer drops from 512 entries to 256 for this 1000-branch hot spot. One aberration here is that the 1024-entry configuration detects a slightly different hot spot that represents a slightly larger percentage of execution, but suffers many more conflicts while detecting it. In summary, for this application and input, a buffer on the order of 512 entries appears to be sufficient. However, a more detailed study of cutting edge applications on a target architecture is needed to properly size the buffer.

CHAPTER 5

REGION EXTRACTION

After hot spot detection, the Branch Behavior Buffer contains a set of frequently executed branches whose blocks constitute a large fraction of the current overall instruction execution. Figure 5.1(a) depicts branch entries stored in the profiling table as circles. The branch execution and direction weights were accumulated during the profiling process to provide behavior information about the branches during the current execution phase. The hot branches and directions are solid-filled and comprise a skeleton of the frequently executed region of code. The next task in the run-time optimization process is to use the skeleton to determine the instruction blocks associated with the branches, as shown in Figure 5.1(b), and to construct a new set of blocks covering the frequently executed ones, as shown in Figure 5.1(c).

The Trace Generation Unit (TGU) is a hardware mechanism for extracting the traces whose skeletons are recorded in the Branch Behavior Buffer. Since the skeleton is obtained from the real streams of execution, the traces often cross-cut multiple functions and modules, thus identifying the important program paths. Likewise, since the skeleton is obtained from a historical record of real execution, the TGU is able to produce traces that have a high probability of future execution. While a compiler or off-line optimizer could perform control-flow analysis over an entire code region to identify and extract traces, a hardware mechanism can produce the traces

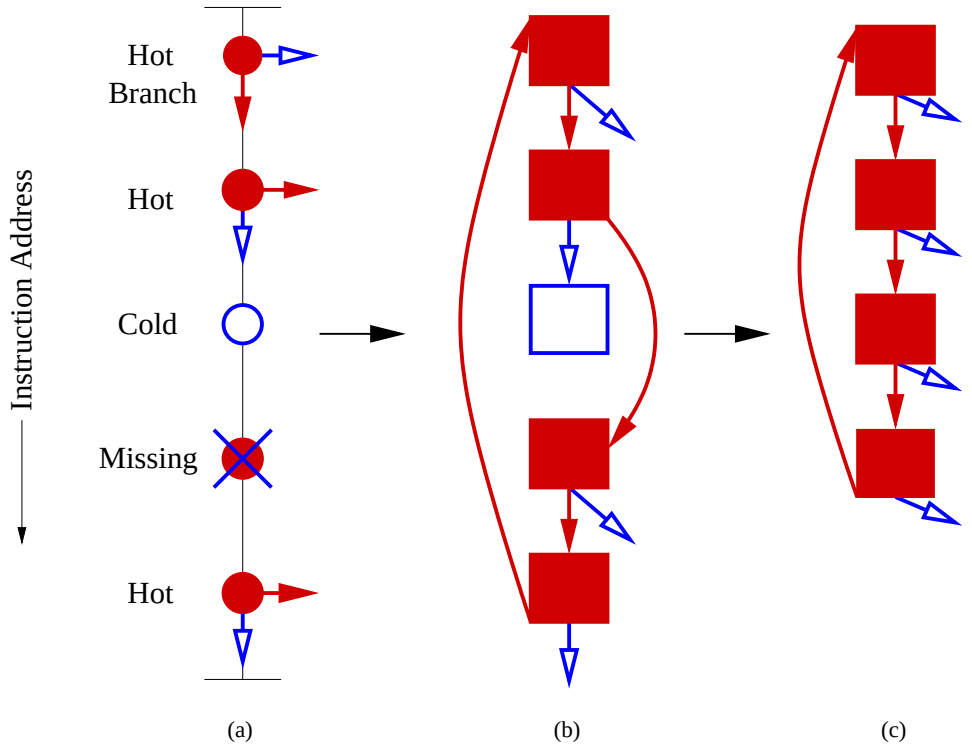


Figure 5.1 Process for using branch profile information to form traces.

with almost zero overhead and latency. This low-cost approach is possible because of the tight integration between the TGU and the Hot Spot Detector.

Traces have been shown to be an effective platform for run-time optimization due to their moderate capacity for optimization but simplicity of analysis [18]. Therefore, a run-time optimizing system can be enhanced several ways by employing a mechanism such as the Trace Generation Unit. First, as previously mentioned, the TGU incurs minimal overhead by performing trace extraction in parallel with the normal execution of the application. This parallelism is possible because the profile that guides the formation process is stored in an easily accessible, dedicated hardware table (the BBB) instead of in memory or in cache. Second, code

straightening optimization is applied as a by-product of the trace generation process, which itself improves fetch performance in the microprocessor. Third, the traces represent the frequent paths through the application that can be subsequently transformed by an optimizer to improve overall application performance. Amdahl's Law states that the overall benefit due to an enhancement is limited by the portion of execution that can benefit from the enhancement [19]. For a run-time trace-based system, this means that both code coverage (fraction of code that can benefit from enhancement) and trace length (unit of optimization) are crucial factors when assembling a fertile code collection for optimization. Fourth, since the traces are restricted to include just the frequent traces, the mechanism filters out spurious traces that when optimized will only provide minimal benefit because they will be rarely executed. Other proposed systems tend to form a great multitude of traces that may bog down the optimizer [20] (described in Section 7.3) and may often displace the important traces in trace repository. However, the TGU will have more difficulty in extracting traces from applications that have little path execution consistency. Last, the TGU extracts the traces and forms them into a collection in memory. The collection is stored in memory in traditional sequential program form that can be fetched by the processor with a traditional fetch mechanism. In this form, branches in traces link to other traces, even forming loops out of traces. While this representation allows for trace persistence, potentially throughout the entire lifetime of the application, it does not provide quite the level of the aggressive loop unrolling inherent to trace cache based systems [21]. Such systems can unroll a loop until the trace cache line is full, thus potentially enabling optimization over a great number of iterations. A typical set of traces may contain one trace with loop preheader code followed by several unrolled loop bodies and another trace with just unrolled bodies. Trace

caches effectively have back edges since optimization is not performed between traces. Here, the first effective back edge depends on the number of loop bodies that fit after the preheader code, and subsequent effective back edges depend on the number of loop bodies that can fit into a trace. To conserve code size inside the TGU-generated loops, however, the amount of unrolling is limited and a back edge from the bottom of the unrolled loop body to the top still remains. The TGU has an advantage in that a loop with a short trip count may be perfectly unrolled and optimized whereas it may be split into two traces inside the trace cache, depending on the amount of preheader code.

The Trace Generation Unit is tightly coupled with the retirement stage of the processor pipeline and with the Hot Spot Detector for maximum efficiency. The mechanism watches the sequences of subsequently retired instructions and constructs traces from those sequences that conform to the bounds of the detected hot spot. The TGU must be able to form good quality traces even though some of the important branches may be missing from the table. Because the detector can profile only a finite number of branches, a few important branches can be missing from the profile due to contention for table entries. This mechanism also adds additional system requirements: an expanded BBB, some associated registers and control logic, and a few pages of reserved virtual address space for each process. Like the BBB, the TGU is not sensitive to latency (it may lag behind actual instruction retirement) and should have little effect on the processor's critical path.

5.1 Overview

The TGU remains idle until the hot spot detection hardware detects a suitable program hot spot. Following detection, the TGU is enabled for a short period of time, during which it constructs a set of traces from the stream of instructions retired by the processor. The TGU spends most of this time operating in a passive mode, scanning the retired instructions for branches that match those captured by the BBB during profiling. The TGU actively generates traces in short bursts (experiments show 0.005% of the total dynamic instructions), writing the instructions to a code cache that resides in the virtual memory space of the process being optimized. Other than a potential slowdown during this active phase of trace generation, the TGU is nonintrusive to program execution. Because virtual memory is used to contain the optimized code, standard paging and instruction caching mechanisms allow translations to persist across context switches. The code cache pages can be allocated by the operating system at process initialization time and marked as read-only executable code.

The allocation of dynamic optimization components to hardware or software affects the overhead, flexibility, and optimization frequency of the resulting system along with the size and shape of the unit of optimization. For example, the larger the overhead, the longer an optimization must persist to amortize its cost and claim a benefit. Hardware mechanisms promise to control the run-time overheads of dynamic optimization systems by transparently applying profiling and optimization techniques. By using dedicated components, hardware mechanisms typically allow for greater parallelism, often processing in parallel with the running application. Software systems typically implement detailed profiling by utilizing instruction interpretation,

or by inserting profiling probes through just-in-time compilation. This leads to frequent transitions between the application, optimizer, and operating system, potentially adding a significant overhead. Infrequent sample-based profiling usually incurs a few percent overhead, while full interpretation is often orders-of-magnitude slower.

Experiments have shown minimal overhead due to the hot spot detection and trace generation process. The HSD can continuously monitor program execution at low cost, conducting profiling without degrading the performance of program until a hot spot is detected. The TGU operates at low cost because it is also a dedicated hardware structure that simply monitors the retired instructions. Some overhead may be incurred, however, while a trace is formed because the construction throughput is likely to be less than the pipeline retirement throughput. Enhancing the Trace Generation Unit to employ optimization techniques is likely to add minimal overhead as well, since the hardware optimizer will operate in parallel with trace generation and native program execution. In essence, ROAR enables full-speed native execution of the application with minimal, decisive, and surgical optimizations to the code. This combination of continuous profiling and precision allows for a faster response than a software system.

One primary benefit of software reoptimization approaches is their flexibility. Within any profiling and code deployment system, a number of strategies can be employed that can dynamically decide when and what to optimize. While the Hot Spot Detector and Trace Generation Unit are hardware structures, they too contain a number of parameters that can be adjusted dynamically. For example, Hot Spot Detector thresholds and timers can be adjusted to vary the code region size detected. The detector could be configured to identify and optimize critical code first, later broadening its scope to optimize remaining code second.

It is often difficult to quantify the different trade-off options between hardware and software because each highly depends on system goals and available mechanisms with associated overheads. While ROAR provides a mechanism with a fast response for smaller units of optimization, a staged approach might work best. Such an approach allows a low-overhead system to make moderate improvements application-wide while, over time, further identifying the very hot regions, handing them off to more aggressive systems with higher overheads that can optimize across entire regions.

5.2 Code Deployment

Because of code self-checks, a criterion for ROAR is that the *original code cannot be altered in any way*. Therefore, any optimizations performed on the program are only performed on the generated hot spot traces. For the same reason, a seamless mechanism for transferring execution into the code cache, instead of changing the branch targets in the original code, is needed. The Branch Target Buffer (BTB) can be used to facilitate control transfers. In some implementations, this structure associates a branch instruction with its taken target by storing the branch's target address. Similarly, ROAR utilizes the address field in the BTB to store the taken target's location *in the code cache*. Therefore, all entry point transitions in ROAR must occur along taken branch paths. Traces begin with the instructions at the target of the branch and can only be reached through the particular transitioning branch.

Other BTB implementations associate a fetch address with the next fetch address. In such a structure, the code cache address of the target could be associated with a fetch address that

would fetch a transition branch. However, this association is not a guaranteed mapping, like a single branch to its target, because there may be many branches with different targets that could be executed from a single fetch. Therefore, this style BTB will predict an entry into the code cache that must be verified when the transition branch executes and, thus, a transition table separate from the BTB would be required. In typical operation, the BTB would be updated with the code cache mapping after the first execution of the transition branch because the BTB would have mispredicted to original code. For the purposes of the remainder of this discussion, the first BTB implementation will be assumed with the understanding that only modest changes would be needed for alternate BTB styles.

After each new trace is constructed, an entry point record for the trace is written to a list located in the first page of the code cache. The record associates the entry point branch in the original code with its target placed in the code cache. During the trace generation process, a timer signals the end of trace generation for the current hot spot. At that time, a routine is initiated to install the list of entry points into the BTB. For each entry point, the BTB target for the entry point branch is updated with the address of the entry point target in the code cache. An *entry point bit* is also set in the BTB to lock the entry in place until a BTB flush. After a context switch, the same routine can be invoked to reinstall the entry points into the BTB on a per-process basis. No new hardware is required, other than that needed to update BTB entries and to ignore branch address calculations selectively for branches that have the entry point bit set.

Self-modifying code (code that writes into its own code segment) presents a challenge to all dynamic optimization systems. Modifications made to the original code segment must also be

reflected in the optimized traces. To prevent optimized code from becoming inconsistent with the behavior of the modified original code, either instructions that have been modified must be updated or traces that contain such instructions must be flushed. Since it is not generally possible to locate all modified instruction locations in the optimized code, many systems (Dynamo [22], for example) flush their entire cache contents when self-modification is detected. Other systems, such as the execution trace cache in the Pentium 4 microprocessor, include a bit for each trace in each translation look-aside buffer entry that is set to true if the trace contains at least one instruction from that page [23]. Thus, when an code page is modified, all traces that contain any instructions from that page are flushed. While this approach is still conservative, it reduces the number of traces that must be flushed. ROAR currently flushes the contents of the code cache and returns to unoptimized code like Dynamo, although each hot spot could contain a list of pages of included instructions. However, all write operations to code pages would have to be trapped and the target pages compared against all existing hot spot lists to check for invalidation.

5.3 Trace Generation Unit Architecture

As the TGU writes instructions into the code cache, it performs two important functions. First, it creates connected regions of code that embody the detected hot spot and defines entry points to those regions. It does this in such a way that if program control enters a hot spot region at a selected entry point, control will likely remain inside the region for a significant

length of time. Second, the TGU automatically performs code straightening along the most frequently executed paths.

The process of copying an instruction into the code cache is referred to as *remapping* the instruction. If the copied instruction is a branch, this process can involve changing the target and possibly the condition of the copy within the code cache. Thus *remapped instruction* or *remapped branch* refers to an instruction within the code cache. If a branch has been *remapped in its taken direction*, then an inverted instance (inverse condition with swapped taken and fall-through targets) of the same static branch has been placed into the code cache. The new taken target may point to the fall-through instructions in the original code, or to another trace in the code cache for the fall-through path.

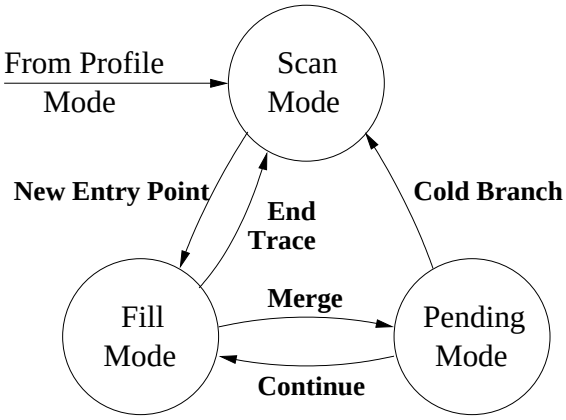
The collection of traces created for each hot spot is called a *trace set*. Although an individual trace may contain internal branches as well as branches to other traces in the same set, it never transfers control directly to code in a different trace set. Therefore, traces generated from a hot spot form a self-contained region of code that can be independently optimized, deployed, and removed if necessary.

To assist on-the-fly trace formation, additional fields are incorporated into the BBB, as shown shaded in gray in Figure 4.1. The code cache taken address and fall-through address fields are used to hold offsets into the code cache at which code following the corresponding branch direction has been placed. Storing the code cache addresses for the branch targets in the BBB allows the TGU to link important branches from a trace directly to the target of a previously remapped instruction in the same trace set. Valid bits for each target indicate whether or not that path has already been generated. By marking the paths that have already been

Table 5.1 TGU registers used for code remapping.

CodeCacheOffset	Offset to next available memory location in code cache. Updated each time the current trace is filled with an instruction.
CurrEntryBranch	Original address of branch instruction used as the entry point to the current trace.
CurrEntryTarget	Code cache target of *CurrEntryBranch*.
PendingTarget	Original address of the instruction that that follows the current trace. Used in Pending mode to determine when to continue filling a trace.
ConsecutiveOffPathBranches	Number of noncandidate branches executed consecutively while filling the current trace.
TotalOffPathBranches	Number of total noncandidate branches executed while filling the current trace.
NextCallID	Integer value for the next unused call ID. A call ID estimates calling context.
CallIDStack	Stack representing the current call-chain during Fill mode. Each entry contains a call ID and a return address. The top entry is for the current function, below it are entries for a fixed number of parents.
ContainsPatchTarget	Single bit that is set when another trace contains a branch that, through a patch, links into the trace being constructed.

generated, the TGU avoids creating redundant traces. A *callID* field also aids trace generation by tagging the target fields to a particular calling context. This prevents linking code from different contexts together, a problem that is discussed in Section 5.4.5. Finally, a *touched* bit is added to support the backtracking operation described in Section 5.4. In addition to the BBB modifications, the TGU uses the set of registers listed in Table 5.1.



(a)

Rule	Condition	Rule	Condition
New Entry Point Transition:		**Merge Transition:**	
1	Executed suitable entry−point branch	6	Executed direction of candidate branch already placed, but other direction not placed
End Trace Transition:		**Continue Transition:**	
2	Both paths from executed candidate branch already placed	7	Executed candidate branch matches pending target
3	Executed non−candidate branch and maximum allowed off−path branches exceeded	**Cold Branch Transition:**	
4	Executed return target mismatches call site	8	Executed non−candidate branch
5	Executed candidate branch will link to recursive context		

(b)

Figure 5.2 Trace generation modes with rules for mode-altering transitions.

5.3.1 Trace generation control logic

This section describes the operation of the state machine in Figure 5.2(a) that controls trace generation. More precise treatment of the Trace Generation algorithm is provided in [3]. Figure 5.2(b) illustrates the decision rules used for each transition arc in the state machine. The trace formation process consists of the following four modes of operation:

- Profile Mode: Search for and detect a hot spot.
- Scan Mode: Search for a trace entry point. Initial mode following detection.
- Fill Mode: Construct a trace by writing retired instructions into the code cache.
- Pending Mode: Pause trace construction until a new path is executed.

When the Hot Spot Detector signals that a new hot spot is available, the TGU transitions from its idle Profile Mode to Scan Mode. In Scan Mode, the TGU performs a BBB lookup for each retired taken branch instruction. If the retired branch is listed in the BBB as a candidate branch that has not been used in a trace, the TGU initiates a new trace. This is accomplished by setting the current trace entry point to the next available code cache offset, storing this offset in the code cache taken field in the BBB, and transitioning to Fill Mode (transition Rule 1). Table 5.2 depicts these actions and transitions taken based on the BBB field contents. For example, the first row represents the transition to Fill Mode when a suitable entry point is found (Rule 1).

During Fill Mode, the TGU writes each retired instruction sequentially into the code cache. All nonbranching instructions are written without modification (default fill rule). However, branching instructions require further treatment. The TGU ignores unconditional jump instructions because the block at a jump target will be filled into sequential locations immediately following the copy of the jump's predecessor. Although conditional branch instructions are written to the code cache, the TGU may invert the branch sense if execution proceeds along the taken path. Conditional branches cause the TGU to perform a BBB lookup to determine

Table 5.2 TGU actions based upon state and BBB entry contents.

TGU Mode	Instruction Type	Candidate Branch	Calling Context	Executed Direction Already Remapped	Other Direction Already Remapped	**Transition Rule**	Next State	Action
Scan	Taken branch or jump	Yes	N/A	No	No	**1**	Fill	Record entry point location in BBB.
	Default					**Default**	Scan	
Fill	Conditional branch	No, and max off-path brs exceeded	N/A	N/A	N/A	**3**	Scan	End trace by branching to original code.
		Yes	Recursive	N/A	N/A	**5**	Scan	End trace by branching to original code.
			Different	N/A	N/A	**9**	Fill	Place inst. Overwrite code cache target location in BBB.
			Same	No	N/A	**10**	Fill	Place inst. Record code cache target location in BBB.
				Yes	No	**6**	Pending	Link trace to code cache target Set pending target to other direction.
					Yes	**2**	Scan	End trace by linking to code cache targets.
	Mismatched return	N/A	N/A	N/A	N/A	**4**	Scan	End trace with return.
	Default					**Default**	Fill	Place inst.
Pending	Conditional branch	Yes	N/A	Matches pending target		**7**	Fill	
		No	N/A	N/A	N/A	**8**	Scan	End trace by branching to original code.
	Default					**Default**	Pending	

how trace generation should proceed. Discussion of filling through calls, returns, and indirect jumps is deferred until Subsection 5.4.5.

If the BBB lookup fails to locate a candidate branch entry, the TGU increments a small counter that indicates the number of sequential *off-path* branches that have been retired. When this counter exceeds a preset threshold (Rule 3), the TGU signals an End Trace condition

and transitions back to Scan Mode. Otherwise, the TGU continues to fill instructions along the executed path. For a reasonably sized BBB, a maximum off-path branch threshold of one allows for an occasional missing branch entry while preventing excessive trace generation down cold paths. Two off-path branch counters are actually used by the TGU. The first limits consecutive off-path branches and the second limits total off-path branches.

When the BBB lookup returns a candidate branch entry, the TGU checks whether the branch has been remapped in the current direction. In the case that the retired branch was taken, the TGU checks the code cache taken field in the BBB entry; otherwise, it checks the fall-through field. If the retired branch has not yet been remapped in the current direction, the TGU continues to fill instructions from the executed path (Rule 9). Before proceeding, however, the TGU marks the remapped target field valid and stores the current trace address into the remapped address field. When emitting the conditional branch, the TGU uses the remapped address from the BBB for the opposite direction if it is valid. This reduces the number of exit points from the code cache to the original code.

If the BBB lookup reveals that the retired branch has already been remapped in the current direction, the TGU stops filling the trace. If the branch has also been remapped in the opposite direction (Rule 2), then the TGU signals an End Trace condition. At an End Trace condition, the TGU writes the trace's entry point to the code cache for future insertion into the BTB and transitions back to Scan Mode. To close the trace, the TGU emits an unconditional jump using the remapped address for the direction opposite the retired branch direction as the jump target.

If the retired branch's executed direction has been remapped, but not its opposite direction (Rule 6), the TGU signals a Merge condition. At a Merge condition, the current branch is

placed in the code cache so that its taken direction links to the already remapped code. Then, the TGU sets the *pending target* register to the target address of the branch direction that has not yet been remapped and transitions to Pending Mode. In Pending Mode, the TGU monitors retired conditional branches. If the TGU encounters a retired branch whose target has not been remapped, it compares the branch's target address with the pending target register. If both target addresses are equal (Rule 7), the TGU signals a Continue condition and transitions back to Fill Mode.

By operating in Pending Mode rather than ending a trace, the TGU forms longer, more complete traces that extend beyond loops. Consider forming a trace that contains an inner loop. When the TGU reaches the loop back edge, it enters Pending Mode, because the executed direction of the branch already has a valid remapped target to the top of the loop. When the control finally exits the loop, the TGU continues filling the trace where it left off.

While the TGU operates in Pending Mode, it is possible for program execution to leave the code that has already been encountered without reaching the pending target address. The TGU easily recovers from this situation by exiting Pending Mode if it encounters a noncandidate or cold branch (Rule 8).

5.3.2 Trace validity

The traces generated by the TGU are identical to the original code with the exception of control flow instructions. The TGU ensures the correctness of all modified branch instructions because it never emits a branch without providing a valid target address. For each branch in a trace, the TGU either uses the original target address of the branch, or it uses a remapped

address that points to a target previously placed into the code cache, which is equivalent to the original code.

The TGU also guarantees that all traces are *well-formed* before the processor can begin executing them. A trace is well-formed if all possible execution paths starting from the trace entry point are valid. Recall that no entry points for a trace set are installed into the BTB until all traces in the set have been generated. Furthermore, traces from one set are never linked to traces in another set. Therefore, as long as the current trace is closed before trace generation ends, all traces in the set will be well-formed before they are installed. Note that trace generation can be terminated by an asynchronous event such as a context switch without detriment. The TGU simply emits a single closing jump instruction if it is interrupted while filling a trace.

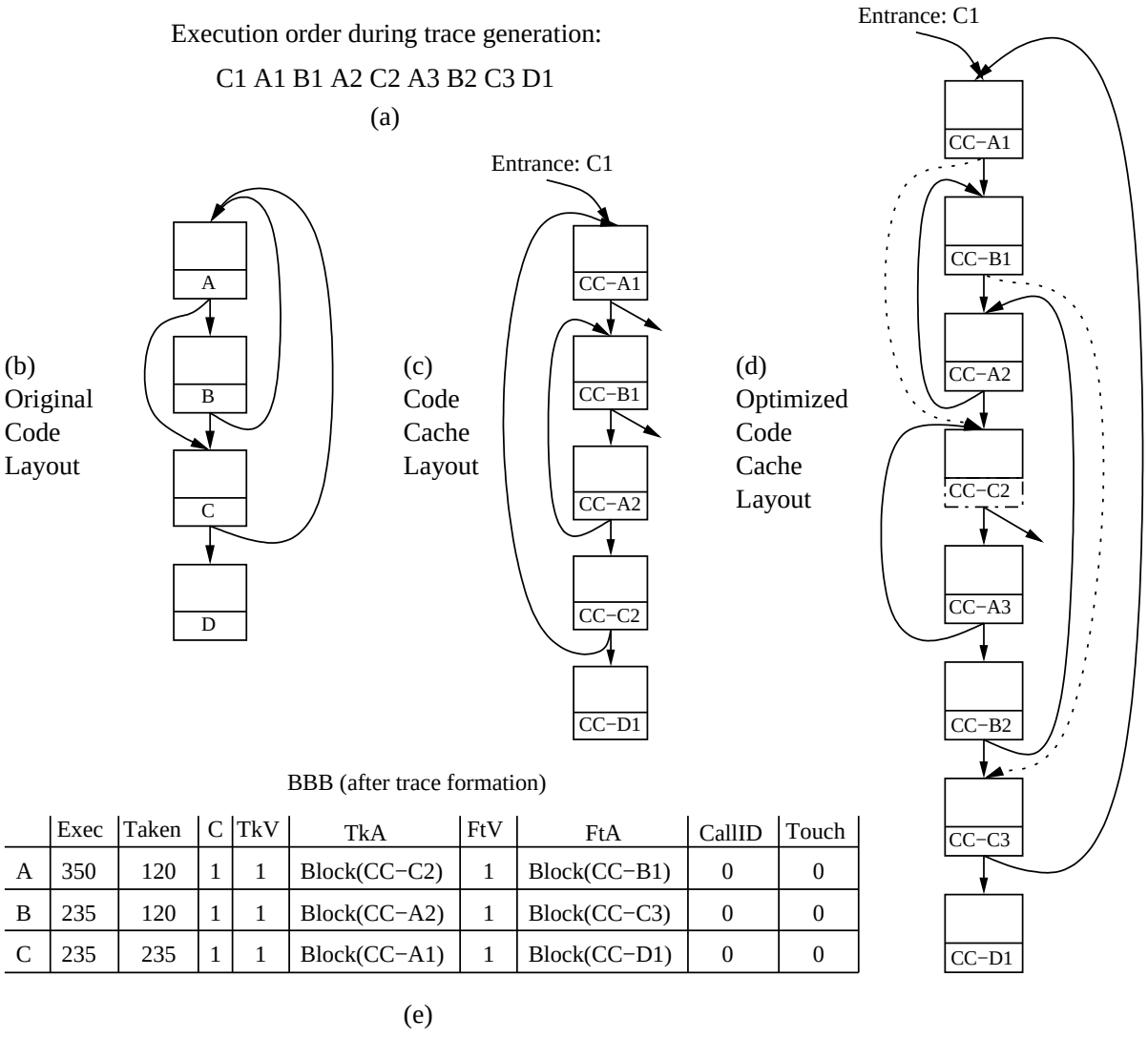
During Fill Mode or Pending Mode it is sometimes desirable for execution to remain in the original code without jumping into the code cache. This prevents new traces from containing copies of code cache instructions and ensures that all exit branches return to the original code. Therefore, while operating in these modes, the processor must ignore BTB entries with the entry point bit set. The time spent in Fill Mode and Pending Mode is small enough but not necessarily inconsequential, and can reduce overall benefits because unoptimized original code executes during all detection processes. This model produces independent hot spots that potentially contain a considerable amount of overlap. From the *134.perl* example in Figure 3.4, block B is the only block in Hot Spot 2 that is not present in Hot Spot 1, yet including a customizable copy in Hot Spot 2 is desirable. In the opposite model, extracted regions for all previously detected hot spots are left to execute, but are not considering during the profiling

and generation processes. This produces a layered effect of hot spots where secondary hot spots are formed that catch frequent border cases or that may catch an outer loop around primary hot spots that are inner loops. This work takes the first approach where previous entry points are disabled during detection. However, a hybrid approach is taken in the optimization section that disables entry points for previously extracted hot spots that are not performing well (i.e., short runs or low frequencies).

5.3.3 Trace generation example

This section provides an example of the trace generation process. Figure 5.3(b) shows the original code layout. The label at the end of each block of instructions represents the static branch instruction that terminates the block. Figure 5.3(a) lists the execution sequence seen after entering Scan Mode. The number following each branch label signifies a dynamic occurrence of that branch. The basic trace generation mechanism described in the previous section generates the trace shown in Figure 5.3(c). The static branches in the code cache are denoted by "CC" followed by the label for the dynamic branch that caused the trace branch to be emitted. The application of two fetch optimizations, *patching* and *branch replication*, results in the trace shown in Figure 5.3(d). Patching reduces premature trace exits while branch replication performs more aggressive code straightening, unrolling loops in the process. Figure 5.3(e) depicts the final contents of the BBB after trace generation.

The branches A, B, and C are all candidate branches and, therefore, potential entry points. However, C is most likely to be selected as the entry point because it is the most frequently taken branch. After detecting a new trace entry point at C, the TGU remains in Fill Mode until



BBB (after trace formation)

	Exec	Taken	C	TkV	TkA	FtV	FtA	CallID	Touch
A	350	120	1	1	Block(CC−C2)	1	Block(CC−B1)	0	0
B	235	120	1	1	Block(CC−A2)	1	Block(CC−C3)	0	0
C	235	235	1	1	Block(CC−A1)	1	Block(CC−D1)	0	0

(e)

Figure 5.3 Trace generation example.

it reaches dynamic branch $C2$. Because the taken target of $C2$ has already been remapped, the TGU transitions to Pending Mode. During this time, the processor executes one loop iteration before reaching $C3$. Execution of the fall-through path of C signals a Continue condition, and the TGU reenters Fill Mode. The TGU continues generating a trace until the number of

sequential noncandidate branches (such as D) exceeds the off-path threshold (Rule 3). The TGU then closes the trace and returns to Scan Mode to search for another trace.

5.3.4 Trace generation parameters

The TGU employs a number of algorithms that require configuration through parameter settings. Most notably, a taken percentage threshold is used to determine whether a branch direction is hot or cold. These classifications are examined during the entry point selection process to choose hot starting points and can be used during the relayout process to detect execution streams that differ from the detected hot spot. In this work, any branch direction with an execution percentages less than 25% is considered cold, and therefore will not be used as an entry point and will be considered out of the hot spot. Note that a 50% taken branch, for example, would be considered hot in both directions. This minimum percentage would likely be set between 10% and 50%.

Another parameter limits the number of consecutive off-path branches encountered before a trace is ended due to Rule 3. Because of the relatively large BBB size used for the experiments, few branches are expected to be missing from the table, and thus two are allowed before Rule 3 is invoked. Similarly, once any trace contains more than four total off-path branches, the trace is terminated.

Last, a time limit is placed on the trace generation process. More precisely, one clear interval number of branches are allowed to retire during the trace construction process for a hot spot. The rationale behind such a time limit is that since a maximum of one clear interval was

allowed to detect a hot spot, then the same interval should be sufficient to perform the trace generation. This is one simple heuristic, but may others are possible.

5.4 Trace Generation Unit Architecture Enhancements

A number additional features have been included in the design of the TGU. The patching optimization is designed to link traces together to increase the percent of dynamic instructions executed from code cache. Branch replication (required for loop unrolling), branch promotion, and automatic inlining are designed to increase trace length to improve front-end useful fetch bandwidth by restructuring code to eliminate some taken control flow instructions. Last, backtracking is used to throw away traces or portions of traces when execution deviates from the profile stored in the BBB. This will allow a fresh trace formation attempt when execution returns to this particular fragment of code.

5.4.1 Patching

Notice from the trace in Figure 5.3(c) that executing the taken path of branch *CC-A1* would cause control to exit the code cache. When this happens, the system executes original code until an installed entry point is encountered. The TGU employs a simple optimization called *patching* to prevent such premature trace exits. When the TGU emits branch *CC-A1*, it places the address of *CC-A1* itself in the taken address field of the BBB entry for branch A, but leaves the field marked invalid. When the TGU encounters branch $A2$, it reads the address of *CC-A1* from the BBB and patches *CC-A1* so that its branch target points directly to the fall-through

path of *CC-A2*, as shown by the dotted line in Figure 5.3(d). Similarly, the TGU patches branch *CC-B1* to the fall-through path of *CC-B2*. In general, patching can be performed in Fill Mode only when the trace encounters a branch whose target has been remapped in the direction opposite from the direction currently executed.

This feature can also benefit from a limiting parameter. Since patching can create a side entry into a trace from a side exit of another trace, the optimization potential of the first trace can be reduced when it is severed into two pieces. One method for limiting the downside is to only patch when the likelihood of executing along the patch to the side entrance is above a set threshold. The likelihood can be calculated by multiplying together the fall-through ratios of all prior branches in the trace from their BBB entries along with the taken ratio of the side exit. High precision is not necessary in the calculation and could be implemented in an inexact way. In this work, at least a 10% side exit percentage is required for a patch to be created.

5.4.2 Branch replication

To improve the useful instruction fetch bandwidth, it is desirable to eliminate as many taken branches as possible without reducing instruction cache performance. *Branch replication* is a general optimization that has the dual effect of both unrolling small loops and tail-duplicating blocks so they exist in multiple traces. Without branch replication, a trace is filled past a particular branch in the same direction only once. Any subsequent copies of that branch in the same trace set are inverted with respect to the first copy such that their target addresses point to the fall-through address of the first copy. Branch replication, on the other hand, allows traces to continue past the branch multiple times without linking back to the first copy of the branch.

Application of this optimization can be seen in Figure 5.3(d). Instead of merging the already remapped taken path of *CC-C2* to Block(CC-A1), *CC-C2* is inverted so that the fall-through path now points to Block(CC-A3). C''s BBB code cache taken target field is not updated since it already contains a valid target. Because the fall-through path of C still has not been seen, the taken path from *CC-C2* must point back to Block(D), which is the fall-through address in the original code. By not updating the branch entry with the new target, some opportunities for linking traces later may be lost; however, this rule allows the algorithm to eventually terminate because only a finite number of side exit locations can spawn new traces. To limit code growth due to unbounded unrolling, a simple heuristic restricts the total size of the unrolled loop body. In this work, the unrolled code is not allowed to extend past a second L1 cache line boundary. Such a boundary was chosen because it was difficult to track the actual number of times a loop was unrolled in a particular instance of the loop, and to better utilize fetch bandwidth by maximally filling the cache line.

The effects of branch replication on branch prediction are likely mixed. Replication of branches will increase the number of predictor entries affected by the branch. Tagged predictors may suffer from increased contention for entries, eliminating other useful branches from the predictor and thus increasing miss rates. Replication of branches also spreads out execution of a single original branch to several copies, thus leading to slower training times and increased start-up miss rates. However, the replication may also lead to increased resolution by allowing more entries to be used to track and predict branch behavior. Increased resolution may lead to improved prediction rates. Realized improvement or detriment depends on many factors including working set size of the predictor, code structure, and variations in behavior that lead to

diverse pattern histories. Effects of replication and extraction in general on branch prediction warrant further study.

5.4.3 Backtracking

The TGU uses a backtracking mechanism to enhance the quality of the traces. Rather than simply filling whichever path executes during Fill Mode, the TGU watches for anomalous behavior. If a branch direction sufficiently deviates from its BBB profile, the current trace is discarded and the processor returns to Scan Mode. If the profiled frequency of the branch's current direction is less than 25% of its profiled execution, then backtrack is initiated. The same hardware that performs the entry threshold check can be used to check the backtrack threshold by shifting the execution counter by two bits rather than one and by using the branch's current direction to select the taken or fall-through profile weight. Suppose that when branch C in Figure 5.3(c) was encountered for the first time, its fall-through path was executed. This would have caused the TGU to return to Scan Mode after resetting CodeCacheOffset to CurrentEntry-Target. Presumably, the loop would be reentered after a short time, and branch A would again be selected as an entry point. When execution eventually does follow the hot path through the loop, the trace will be completed as before.

To correctly support backtracking, some additional considerations must be made. First, if a branch from a previous trace is patched to the current trace, discarding the trace would cause the branch's target to point to invalid memory. This is avoided by setting a single bit (ContainsPatchTarget) whenever a branch from a previous trace is patched and suppressing backtrack if the bit is set. Removing a trace may also create inconsistent BBB entries when

the taken or fall-through fields point to the discarded trace. If, in the future, another copy of this branch is written to a trace, the branch would be patched to an invalid memory location. To handle this problem, each BBB entry contains a touched bit that is set when the entry is updated. During backtracking, the target fields for touched BBB entries are invalidated. Both the touched bits and the ContainsPatchTarget bit are reset whenever a trace is finally committed.

5.4.4 Promotion

High instruction issue rates are often limited by the number of branches that can be predicted in a single cycle. One method for overcoming this limitation is to mark the instruction with a static prediction via a technique called *branch promotion*. Some trace cache implementations use a Branch Bias Table [24] to track the long-term behavior of the branch, promoting consistent instructions in traces so that they do not require a dynamic prediction. When the static prediction is wrong, however, the processor suffers a branch misprediction penalty, and is likely to cause the promoted instruction to revert back to its dynamically predicted form.

Similarly, wide-issue instruction cache mechanisms suffer from the same limit on branches per cycle. However, the hot spot detection mechanism is well-suited to make promotion decisions during remapping, because a profile of the branches exists in the BBB. Since mispredictions are expensive, we chose to promote only instructions that execute 100% in one direction, according to the BBB. In the example in Figure 5.3(c), *CC-C2* can be promoted to a statically predicated branch because its taken and executed counters have the same value. Occasionally, branches change their behavior over the course of program execution, causing mispredictions that may negate much of the benefit of their promotion. Thus, we require a means for demoting

such misbehaving instructions. We propose a small buffer (empirically chosen to contain 16 entries in the current implementation), called the Branch Demotion Buffer, that tracks a typically small number of promoted branches that exhibit mispredictions and marks the misbehaving instructions in the instruction cache to force a dynamic prediction. The buffer is organized as a fully associative cache containing saturating counters. Demotion or repromotion is performed on the instructions when the counter reaches set threshold values.

5.4.5 Automatic call, return, and indirect inlining

Call and return instructions account for a significant portion of control transfers, and benefit can be gained from inlining frequently executed calls. During trace formation, the TGU inlines subroutines that are part of the current hot spot. Inlining an individual call site requires only minor additions to the trace generation logic. However, the TGU should avoid linking different inlined copies of the same original subroutine to each other because each copy may have a different calling context. Consider the Function `A()` that makes two distinct calls to Function `B()` in Figure 5.4(a). When the likely paths are inlined into Function `A()` in Figure 5.4(b), linking the unlikely path from `B-BB1''` to `B-BB2'` could lead to a return to `A-BB2'` instead of `A-BB3'`. Special call and return instructions, described later, will prevent execution from returning to code in the wrong context and will force a return back into original code. Furthermore, the link may break the contiguous trace through B into two regions, thus inhibiting optimization. To limit linking of traces to those with the same calling context, the TGU attaches a call ID to each inlined call site that is unique within the current trace set. The TGU maintains the next assignable call ID in a register, which can be reset upon detection of a new

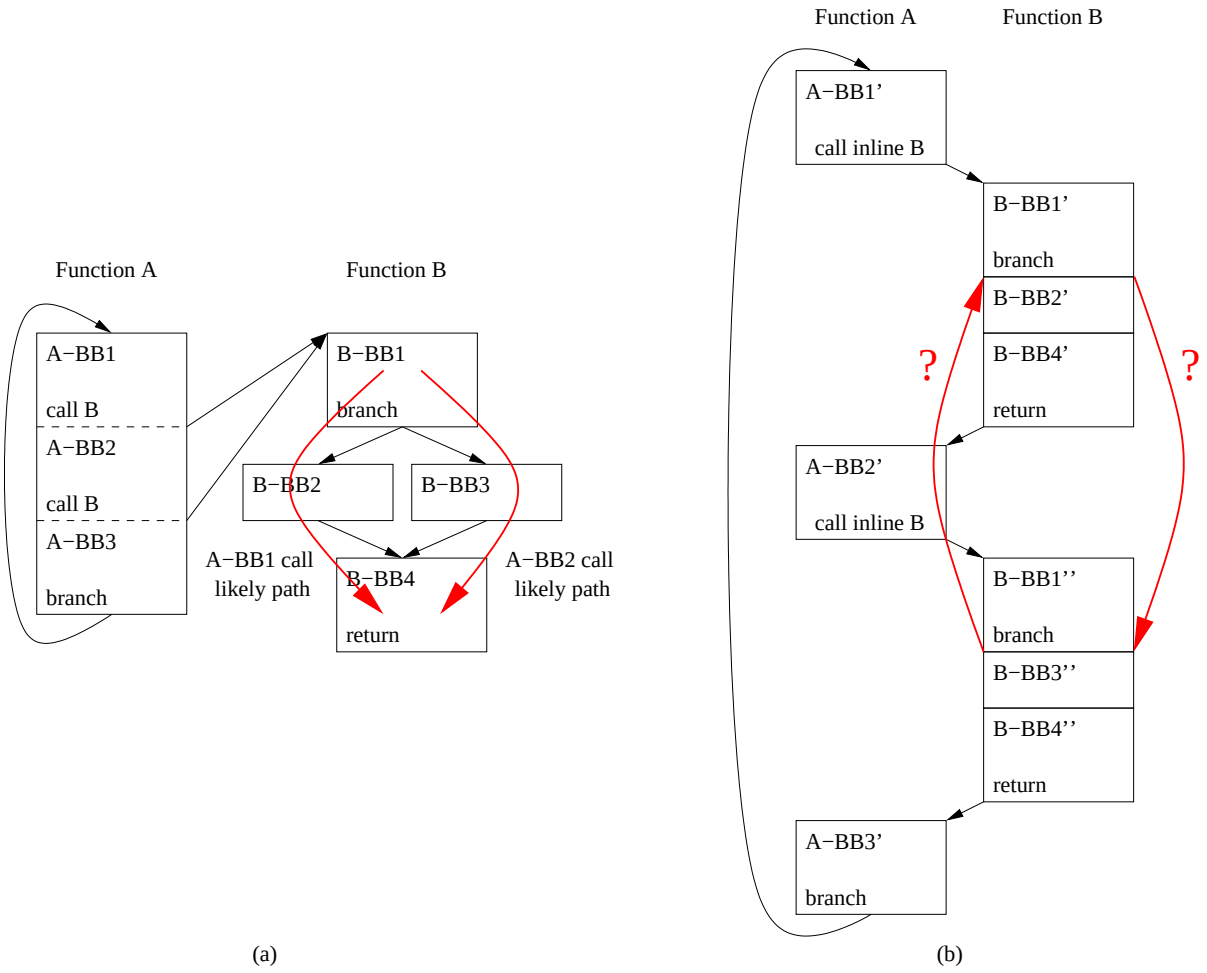


Figure 5.4 Example of maintaining proper calling contexts within a trace. (a) Function `A()` makes two distinct calls to Function `B()` in its loop body. (b) Each call accesses Function `B()` differently and is inlined accordingly. Linking the less likely directions to the other inlined copy could lead to a return into code from a different context.

hot spot. During trace generation, the TGU also maintains a call ID stack that represents the current calling context.

In addition to branch instructions, the TGU identifies call and return instructions during Fill Mode. When the TGU encounters a retired call instruction, it emits a CALL_INLINE instruction, described in detail in Section 5.5. It then pushes a new call ID onto the call ID

stack along with the return address of the call. Similarly, when the TGU encounters a return instruction, it pops the top entry from the call ID stack. If the stack is empty, or if the return address does not match the next retired instruction address, then the TGU signals an End Trace condition (Rule 4). This is usually a result of executing a return instruction without having inlined its corresponding call.

A slight extension to the BBB access performed by the TGU upon retirement of a conditional branch has been made to support inlining. When the TGU updates the remapped target field in the BBB, the TGU also stores the current call ID. Additionally, when the TGU performs a BBB lookup, it compares the recorded call ID with the current call ID. A remapped target is only considered valid if the two call IDs match. This effectively prevents an inlined subroutine from being linked to another copy of the same subroutine in a different calling context. The BBB lookup also compares the call ID field with those on the call ID stack. If the branch entry's call ID is found in a previous stack entry (Rule 5), then a recursive call has been made. In relayout-based approaches that perform aggressive inlining, such as the TGU, deep recursive inlining and relayout must be limited to prevent massive code growth, whereas other approaches may be able to form traces with multiple recursive copies present. In this case, the TGU signals an End Trace condition to avoid recursive inlining.

The Trace Generation Unit is also capable of forming traces across indirect jumps. The new JMP_INDIRECT_INLINE instruction, as will be described in Section 5.5, positions one of its indirect targets immediately following (or in line with) the remapped copy of the jump. Currently, the selection of the target is based upon the target that was first encountered during the trace formation process. However, like indirect branch predictors that make a prediction

from a set of previously seen targets, an extension to the BBB could be constructed to track the weights of each indirect target. Thus, for multitarget jumps, the most or one of the most frequent targets could be selected. One typical use of indirect jumps is in the calling sequence of a shared library function, where there is only a single target of the indirect instruction. This feature enables the aggressive optimization and scheduling across module boundaries.

Dynamic function inlining presents a unique microarchitectural challenge because of its effect on return address prediction. The goal of the trace generation mechanism is to extract and re-lay out the most heavily executed paths in the execution phase. Infrequently executed side exits branch back to original code sequences, thereby limiting the negative instruction cache effects and optimizer workload for unimportant traces. When executing a trace that has been dynamically inlined, no branch prediction is required for the inlined calls since execution simply falls through to the target positioned after the call in the trace. However, execution could leave a trace along any side exit, potentially after several inlined calls, and end up back in original code where the processor state must be consistent with native un-inlined execution. The process stack must be maintained during inlined calls so that the proper return address and local variables are available both inside and outside the trace. Furthermore, while not strictly a requirement, the hardware return-address predictor should also be maintained to ensure continued efficient execution. This predicator is responsible for fetching instructions in the caller code beyond a return in the callee to keep the processor pipeline full. It is updated with the instruction immediately after the call (the return point) each time a call is made. The process stack maintains the real return address and any prediction is always checked for correctness in the execute stage of the pipeline. However, to avoid numerous return address mispredictions

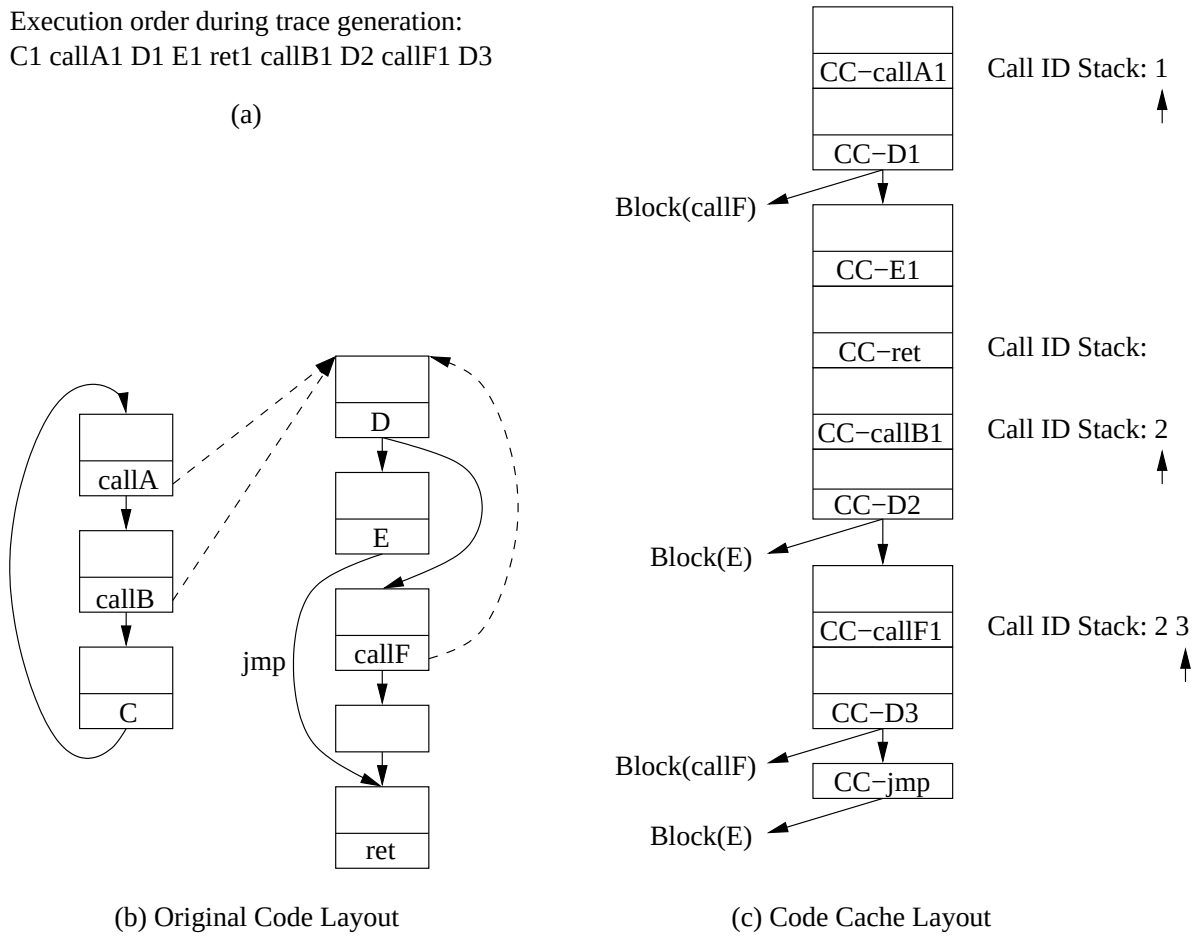


Figure 5.5 Trace example with inlining.

due to returns executed in original code, inlined calls made inside traces must also update this predictor.

5.4.6 Automatic inlining example

Consider the example shown in Figure 5.5. The caller consists of a single block loop that makes two serial calls to the same callee. The trace generation process begins as normal, placing the block terminated by $callA$ (Block(callA)) into the code cache. To inline $callA$,

the TGU pushes the next available call ID (1) onto the call ID stack along with the expected return address (Block(callB)). Next, the TGU inserts a CALL_INLINE instruction in place of the original call. When executed, this instruction saves the return address of the original call site, Block(callB), as the return address, but allows execution to fall through to the next instruction in the trace. Saving an original location as the return address is a safety mechanism that guarantees return of control to the correct function if execution takes an early exit from the trace. This also hides the existence of the code cache from programs that read the return address value directly. Some architectures, such as x86 [25], push the saved return address onto the process stack during a call and pop it off the stack on a return. Others, such as Alpha [26] and Itanium [27], store the return address into a register. On Alpha, explicit code must be included to move the return address into the process stack, while the register stack engine on Itanium may be used to automatically retain the return address for each function call. However, the operation of the CALL_INLINE instruction is essentially the same; the architecture stores the return address of the original call site into the return address register.

As the TGU fills the body of the inlined subroutine (D, E, and "ret"), the BBB entries for D and E are tagged with call ID 1. When the TGU reaches the return instruction, it pops the top entry from the call ID stack and verifies that the expected return address (Block(callB) matches the address of the next retired instruction. The TGU then emits a RETURN_INLINE instruction and continues filling the trace. Although the RETURN_INLINE normally allows control to fall through to the next instruction in the trace, it still must compare the return address found on the process stack (or in the return address register) with the original address of the next trace instruction. This check is necessary in the event that a program directly modifies

its return address. If the comparison fails, the RETURN_INLINE behaves exactly as a normal return instruction.

Following the RETURN_INLINE instruction, a trace is formed through Block(callB), and inlining of $callB$ begins. While inlining the second copy of the subroutine, the call ID stack contains call ID 2. This prevents the TGU from patching the taken target of *CC-D1* to the fall-through target of *CC-D2*.

After Block(callF) is filled, another call site to the same function is encountered. CallF is inlined as before, and the next call ID (3) is pushed onto the call ID stack. Block(CC-D3) is then filled into the trace. When a BBB lookup for $D3$ is performed, the BBB Call ID field value (2) matches one of the call ID stack entries below the current stack pointer. Therefore, the condition for Rule 5 is satisfied, and the TGU ends the trace by emitting a conditional jump to Block(callF) and an unconditional jump to Block(E).

5.5 New Instructions

The following are new instructions that are used within the code cache. These instructions are designed to support run-time optimization in hardware but are not visible to the programmer. Although it might be possible to emulate these operations with traditional instructions, they are proposed to be implemented in the microarchitecture for maximum efficiency.

- CALL_INLINE(return addr to *original* code)

  Unlike a normal function call, the program counter is set to the next sequential instruction, and the return address is set to the *operand value* rather than the next PC.

The process stack (or return address register) and the branch prediction Return Address Stack (RAS) are properly maintained in case a normal return is executed later.

- RETURN_INLINE(expected return addr in original code)

  Execution speculatively continues with the instructions immediately following the RETURN_INLINE. Meanwhile, the operand, which is the expected return address, is compared to the return address on the stack (or in the return address register). If the values do not match, then a misprediction occurs and a normal return is executed.

- CALL_INDIRECT_INLINE(indirect addr, inlined addr, return addr to original code)

  The actual indirect target is calculated normally and compared against the inlined address operand. If there is a mismatch between the actual target and the inlined target, a normal call is made to the calculated target. Otherwise, a CALL_INLINE is performed. In both cases, the original return address provided by an operand is pushed onto the stack (or stored in the return address register).

- JMP_INDIRECT_INLINE(indirect addr, inlined addr)

  The actual indirect target is calculated normally and compared against the inlined address operand. If there is a mismatch between the actual target and the inlined target, a normal jump is made to the calculated target. Otherwise, control passes to the instruction immediately following the inlined jump. This instruction is particularly useful for optimizing across dynamically linked library boundaries.

5.6 Algorithmic Characteristics

The TGU has a number of differences from other trace formation systems and exploits several application characteristics to accomplish its goals. The following sections describe some of these differences and characteristics in relation to TGU output. Specific related work is found in Chapter 7.

5.6.1 Structured trace formation

The Trace Generation Unit extracts traces from the hot spot region in a structured manner. Under this strategy, the TGU builds a copy of the original code with its essential control-flow but uses profile data to straighten the hot paths. Most other trace-based dynamic optimization systems generate sequences of arbitrary blocks from a single instance of execution with little or no profile information. These approaches tend to have a significant amount of code replication because traces can begin at almost any point in the code. Due to the desire for the traces to have high code coverage, new traces often begin right where other traces end, implying many potential trace starting points. For a particular loop, this may mean that a handful of traces exist to cover the loop, each beginning at different points in the loop body. Furthermore, traces that cover loops may contain several unrolled copies of the loop body equivalent to the number of iterations encountered in the instance of the loop when it was created.

Trace replacement policies may also increase effective code replication. Some systems replace a trace with a new one when the current execution flow does not match the flow captured in the trace. Essentially, if an early exit from the trace is taken (due to greater or fewer actual

iterations), the old trace may be discarded in favor of a new one. This policy may not allow enough time for an optimizer to transform the trace before it is discarded, and may overwhelm an optimizer with frequent new traces. Allowing old mismatched traces to remain is also suboptimal as the traces are clearly optimized for a particular iteration count. Execution of fewer iterations than are unrolled within the trace wastes work speculatively performed for the unneeded subsequent iterations, while execution of more iterations than are unrolled limits the iteration overlap to the amount detected when the trace was formed.

As previously mentioned, dynamic optimization mechanisms that form traces or regions across function boundaries must also ensure that all indirect control transfers reach the their correct targets. Considering trace-based mechanisms, one target is often placed immediately after, or inline, with the indirect control transfer. A mechanism is in place that allows control to fall through the indirect instruction if the intended target has placed after the indirect instruction but forces a branch if mismatched. Because many dynamically linked library functions are actually accessed through indirect instructions, the algorithm readily picks one target of an indirect branch or call to inline.

Return instructions are also essentially indirect branches where the target address is located on the process stack or in a return register. Figure 5.6(a) describes the general problem of inlining returns. In this example, the trace was formed from Function `C()` back through `A()`. An optimizer would attempt to speculate instructions from those functions from after the call site up into the end of the callee. However, Function `C()` may be a library function with many possible callers. Execution in this particular trace may cause an attempted return to a mismatched return target, and may have caused a significant amount of unnecessary speculative

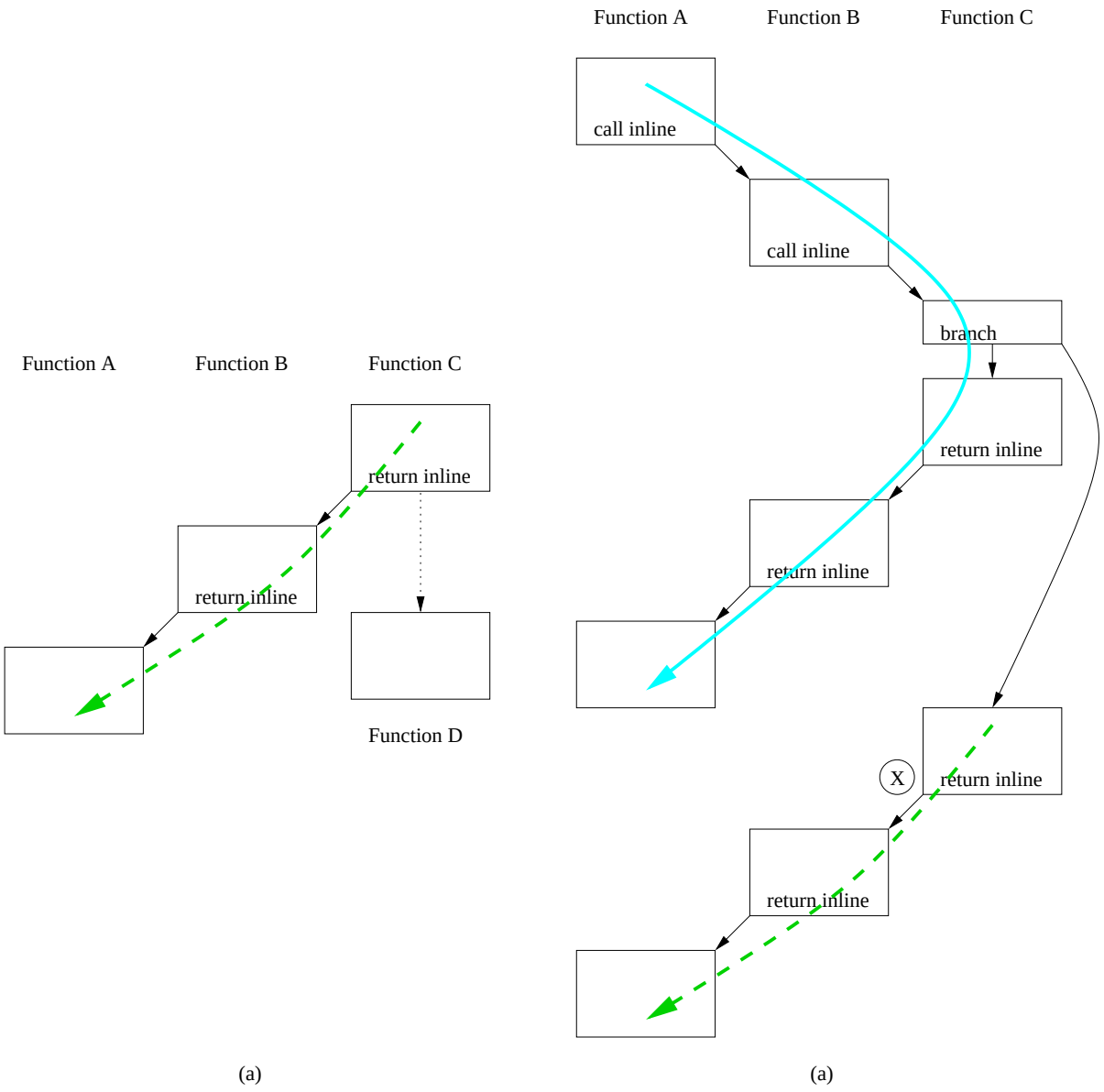


Figure 5.6 Trace formation starting inside callee functions. (a) The trace represents a calling context from Function A to B to C. However, the trace could be entered through an entry point in C, even for a calling context of Function D to C. (b) The main trace (solid arrow) formed with a side exit trace (dashed line) that proceeds through inlined returns even though there are no matching returns in the trace itself.

work for the mismatched target. The mismatched return target would be caught in the execute or memory stage of the processor pipeline when the desired return target is read from the process stack, thus triggering a pipeline flush. This is also true of general mismatched indirect control transfers. Some trace-based systems would then potentially reform the trace to match this particular calling context. However, there is potential for frequent reformation, especially if a particular loop makes two independent calls to the same function yielding two return paths. To avoid this problem, other trace cache-based systems do not form traces beyond returns at all. However, unlike general indirect control transfers, returns to callers are almost always perfectly predictable[1] because matching calls (indirect or not) must have been made to execute into the callees prior to the returns. The structured approach to trace formation, described below, forms a list of the calls made to reach each point in the trace and therefore can reliably inline across returns back into callers by looking at the list. Because the algorithm relies on this characteristic, traces will be ended when a return is reached where the original call was not observed.

The vision described in the algorithm employed by the TGU takes a more structured approach by attempting to inline callee functions into their call sites within the hot spot. Unnecessary control-flow is removed where possible including call, return, unconditional and conditional jumps. By utilizing Pending Mode to wait for the ends of loops, the TGU attempts to form traces that pass through callees back to the caller (trace depicted with a solid line in Figure 5.6(b)). Traces are formed through returns when a matching call is found. According

[1]The most notable exception is `_longjmp()` which will return to an environment and context saved in a previous `_setjmp()`.

to the vision, the TGU also groups trace fragments beginning in callee functions in with the remainder of the extracted hot spot code (trace depicted with a dashed line in Figure 5.6(b)). While this trace does not have matching calls and returns, the returns in the dashed trace are matched with the calls in the solid trace. The only restriction is that the dashed trace can only be entered from the solid trace so that the returns are guaranteed to match the calls. However, this vision is not yet fully realized because the implementation does not currently form fragments starting from side exits in callees that traverse back through returns because the TGU cannot uniquely match the returns with the calls. The implementation currently will form the side traces up to the return where the trace ends (point X in the example) but will make the side trace a normal trace entry point. Like the other trace-based approaches, the return context is checked in the execute pipeline stage, and therefore having the correct calling context inlined into the trace is not absolutely required but is highly desirable.

The TGU relayout optimizations create a larger uninterrupted sequence of instructions for the optimizer to operate over. Loop unrolling is also employed to further lengthen the sequences. However, the TGU traces still include loop back branches that prevent optimization of the first iteration with the loop preheader. Likewise, the loop back branches also limit the number of iterations that are overlapped to the number unrolled, whereas other trace-based approaches may be able to unroll more aggressively but may require many more traces be formed and optimized. Unrolling also has an unfortunate side effect in that the BBB, as previously mentioned, has only one entry to monitor any particular original branch. Unrolling and inlining may, however, cause branch replication, but the BBB is only able to track one of the remapped instances. Future designs of the BBB may be able to use empty fields in the BBB to

form a linked list of branch instances to provide a better platform for linking side exits to new trace heads.

5.6.2 EPIC versus x86

Applications compiled for EPIC machines have slightly different branching characteristics than those compiled for x86 because of the aggressive nature of the compilation process. Since current EPIC processors have in-order cores, the compiler is responsible for aggressive code scheduling. To accomplish this, the compiler profiles an application to gather typical usage patterns of the application which reveal common branch directions and, hence, common execution paths. This information is used to straighten the code which converts some of the taken branches into fall-through branches. The paths that are formed then undergo aggressive scheduling with speculation. Compilers for x86 architectures have been slow to adopt profiling because of the complexity that the process adds to the development cycle, and because the architecture provides little means for expressing aggressively scheduled instructions. EPIC architectures also have increased capability for executing multiple instructions per cycle. Thus, techniques such as function inlining and loop unrolling are more aggressively pursued to increase the scope of the instruction scheduler in order to utilize the available parallelism in the processor.

As previously mentioned, the compiler may have performed many optimizations based solely on compile-time profile information, which may or may not be representative of future usage patterns. When the gathered profile information is correct, ROAR provides limited benefit because it currently performs many of the same optimizations as the compiler: code

straightening, function inlining, instruction scheduling, etc. Furthermore, the ROAR architecture was originally designed to use taken branches as entry points into the code cache. The rationale was to limit required changes to the branch target buffer by overriding a predicted taken branch's target with the code cache target. However, in EPIC codes, the compiler often eliminated a sizable number of the taken branches through code straightening. Eliminating taken branches also has the undesirable effect of eliminating a number of potential entry points leading to delayed transitions and less time spent in the code cache.

5.7 Trace Generation Unit Experimental Setup

Instruction-by-instruction simulations of the TGU were performed on the benchmarks described in Table 4.2 in Section 4.3. The simulated hardware parameters were chosen to model state-of-the-art processor components in order to evaluate the performance contributions made by the TGU. The simulations featured a sequential-block instruction fetch unit with a 64 KB, four-way set associative, 128-byte line, split-block, 10-cycle miss penalty L1 instruction cache (ICache). The L2 ICache consists of a 512 KB, two-way set associative, 256-byte line, split-block, 100-cycle miss penalty cache. Some fetch units were also coupled with trace caches featuring either 128 (8 KB) or 2048 (128 KB), four-way set associative, 64-byte lines. The Branch Behavior Buffer configuration used in these experiments, described in Table 5.3, is predominantly the same as in the Hot Spot Detector evaluation experiments except that a smaller but more associative table is also found to work well. In these experiments, the BBB operated continuously but only tracked instructions outside of the code cache (determined by instruction

Table 5.3 Hardware parameter settings.

Parameter	Setting
Number of Branch Behavior Buffer sets	256
Branch Behavior Buffer associativity	4-way
Executed and taken counter size	9 bits
Candidate branch execution threshold	16
Refresh timer interval	4096 branches
Clear timer interval	32 768 branches
Hot Spot Detector counter size	13 bits
Hot Spot Detector counter increment	2
Hot Spot Detector counter decrement	1

Table 5.4 Fetch mechanism models.

Model	L1 ICache size,way,block	Trace Cache size,way,block,bias table	HSD and TGU
Traditional	64 KB, 4, 128 B	none	no
with TGU	64 KB, 4, 128 B	none	yes
Trace Cache	64 KB, 4, 128 B	8 KB, 4, 64 B, 4 KB	no
TC with TGU	64 KB, 4, 128 B	8 KB, 4, 64 B, 4 KB	yes
Trace Cache	64 KB, 4, 128 B	128 KB, 4, 64 B, 16 KB	no
TC with TGU	64 KB, 4, 128 B	128 KB, 4, 64 B, 16 KB	yes
Traditional	128 KB, 4, 128 B	none	no

address). Table 5.4 summarizes the various processor configurations. The trace caches are allowed to form traces containing instructions from the code cache.

The simulated ICache model has a split-block configuration such that each line is divided into two banks. If a request falls into the second bank, the first bank of the subsequent cache line is also returned, if present. The instruction buffer is capable of delivering up to 16 instructions per cycle to the decoders, but will not issue instructions past a predicted taken branch. Up to three branches may be issued per cycle, and any instructions in the fill buffer that fall after the

third branch will not be used until they are verified to be on the predicted path. The ICache assumes predecode information to identify instruction boundaries and branches.

A 14-bit-history branch predictor called *gshare* [28] is modeled with a pattern history table consisting of entries with seven 2-bit counters, together capable of three predictions per cycle. In addition to the conditional branch predictor, a 32-entry return address stack and a 1024-entry indirect address predictor are provided. We model an ideal BTB to isolate the effect of storing entry points in the BTB. The entry point replacement policy has been deferred for future work.

We also model a trace cache that is indexed on the trace's starting address and allows partial matches (it has the ability to fetch the beginning of a trace up to a prediction mismatch). Both trace cache models (8 KB, 128 KB) are coupled with an ICache and use the same branch predictor as the ICache. When a fetch request is made, both units are accessed in parallel; a trace cache hit always takes precedence over an ICache hit, and only when both caches miss is the L2 ICache accessed. The trace cache is block-based and is modeled after the design in [24]. Each cache line is 64 bytes wide with slots for 16 instructions and up to 3 branches. Four target addresses are stored in the line to provide the next fetch address in case of partial matching. Traces end when the limit on instructions or branches is reached, or when an indirect branch instruction is encountered. The traces are built in basic-block granularity unless more than half of the line will be wasted, in which case partial blocks may be filled. The trace cache also utilizes a Branch Bias Table (BBT) of 1024 or 4096 entries (approximately 4 bytes each) to facilitate branch promotion within traces. Including the additional target addresses and tag stored in each cache line, the combined size for the 8 KB trace cache and 1024-entry BBT is approximately 15 KB.

5.8 Trace Generation Unit Evaluation

In order to evaluate the performance of the Trace Generation Unit, the quality of the generated traces is examined. Because the generated traces will form the *unit of compilation* for the optimizer, it is critical that they cover the frequently executed paths in the hot spot. Furthermore, program performance will benefit from the formation of longer traces because they provide the optimizer with a larger window on which to operate. Optimizations such as partial inlining, loop unrolling, and branch promotion can be utilized to assist in the formation of longer traces. In addition, a trace optimizer may be employed that performs rescheduling and other optimizations on the traces. For this section's results, optimizations that improve instruction fetch performance on a traditional fetch mechanism were employed, and the resulting fetch performance compared to that of a trace cache fetch mechanism.

Figure 5.7 depicts an optimized trace taken from the *130.li* benchmark, as was previously introduced in Figure 3.5. This particular trace highlights the use of partial inlining to form a long trace through a complicated calling sequence. The TGU forms a single trace that begins prior to the call of `evform`, continues following the hot branches while inlining both calls to `xlygetvalue`, and returns to `evform`, where the TGU terminates the trace because the maximum number of allowed off-path branches has been exceeded. This trace contains 284 instructions and has 10 inlined function calls. Its execution accounts for 10% of the optimized fetch cycles of the entire application and achieves 15.1 fetched instructions-per-cycle (FIPC) on a 16 instruction issue architecture. The simulation profiled the execution of the trace, counting the number of trace exits taken. These exits are shown between the blocks in the figure,

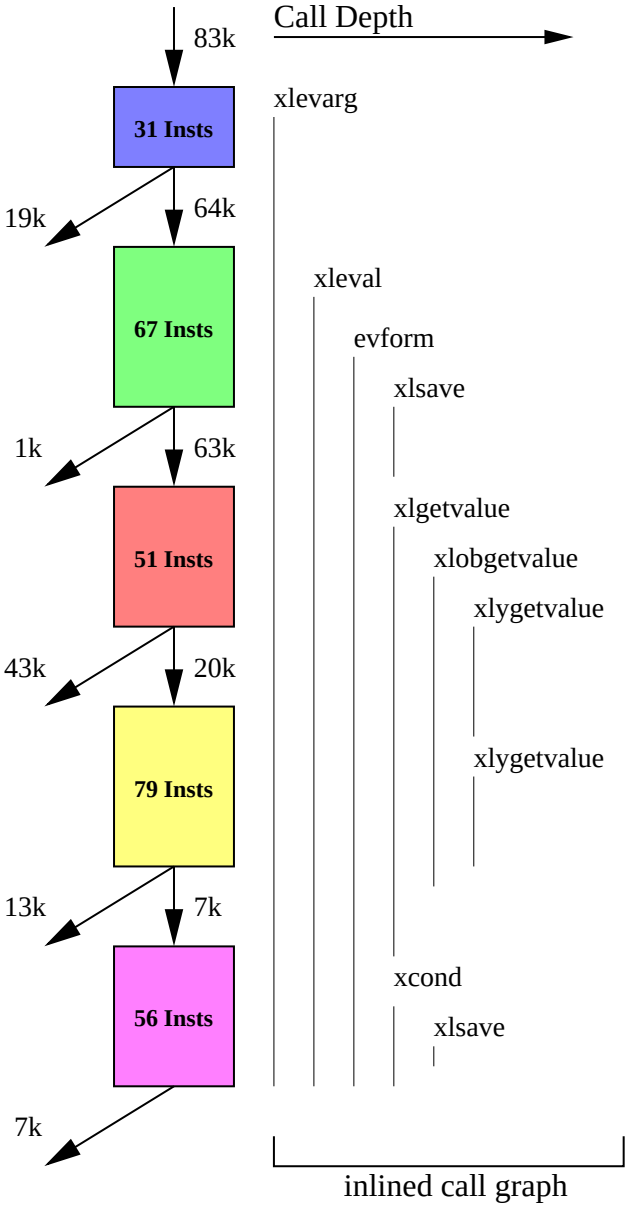


Figure 5.7 Example generated trace from *130.li* optimized for instruction fetch performance.

while a large number of other side exits that are never taken have been eliminated. Clearly, a large number of trace executions proceed at least through the third block, providing evidence that optimizations to reduce the dependence height of that path would increase execution performance. Considering the third branch, the trace appears to have been generated along the less frequently executed path. However, at the time of hot spot detection, that branch flowed 100% of the time to the fourth block. This branch provides evidence that individual branches may also exhibit phased behavior.

Figure 5.8 depicts the execution patterns for six hot spots in the *130.li* application. Unlike the other results presented in this chapter, *130.li* for this example was compiled for an EPIC machine that is described in Section 6.6. For each 100-K instruction time slice, blocks are plotted for the hot spots that execute in the time slice and are shaded darker for higher time slice execution percentages. The execution phases of this application, represented by the various hot spots, are clearly visible in the graph. For instance, hot spot 1 contains functions for reading the input and building new internal representation nodes. Note that after only 4M instructions, this hot spot is no longer active and could be discarded to free up code cache memory for future hot spots. This situation illustrates the modularity that hot spot extraction provides by grouping, optimizing, and managing relevant codes together. Hot spots 2 and 3 contain the two-pass garbage collection process of marking reachable nodes (hot spot 3) and sweeping away unreachable nodes (hot spot 2). Hot spots 4 and 5 contain the evaluation of input functions, and largely contain the same basic evaluation operations. However, each contains several functions not present in the other. When an evaluation function is called which is not present in the current hot spot, execution transitions to the other. Execution may remain in this

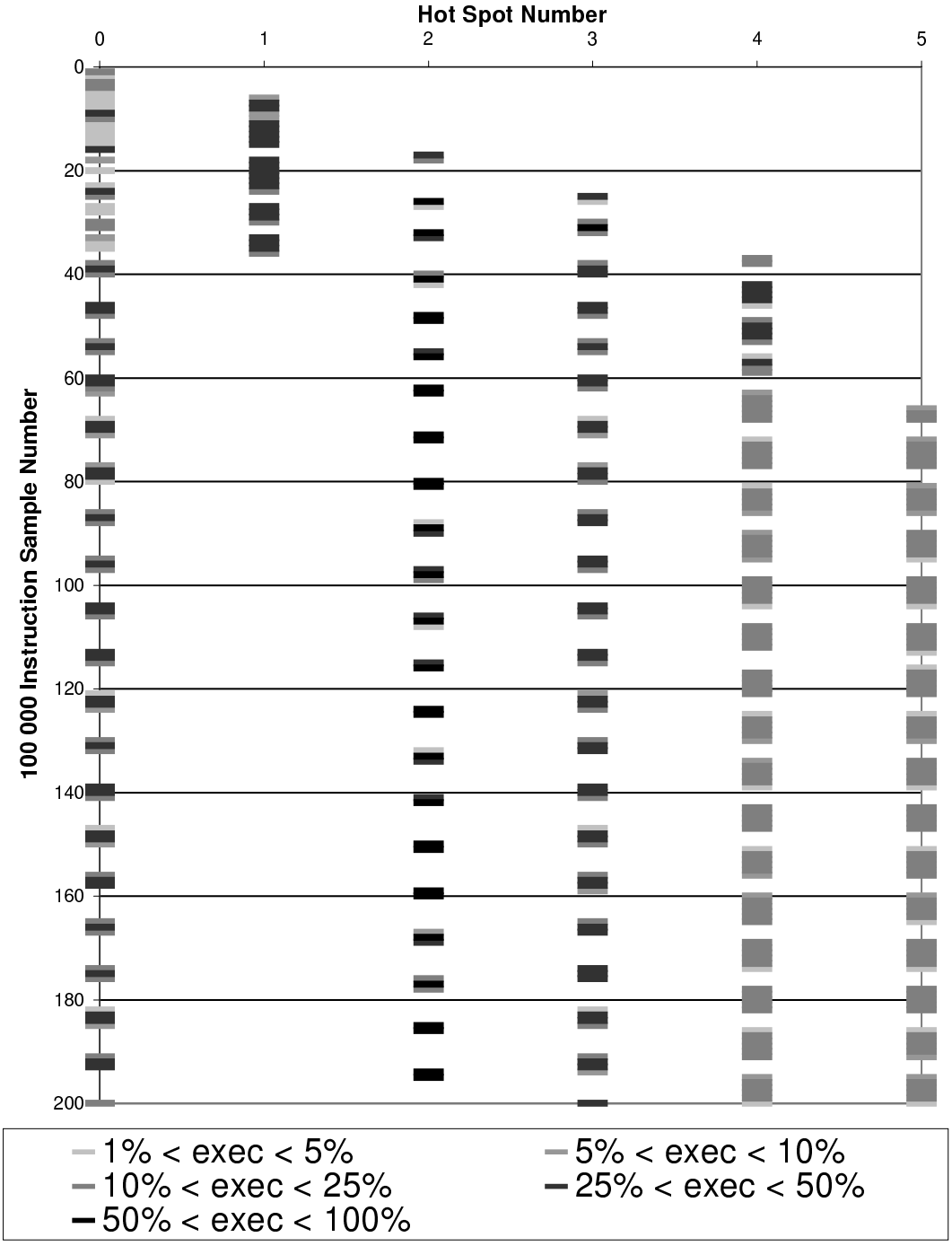


Figure 5.8 Fraction of instructions spent in each hot spot during 100 000 instruction slices from *130.li*.

hot spot because the common evaluation functions are present in both. The transitions clearly occur rather frequently since the execution percentages for each are similar and stable over time. This execution pattern may be better served by a single, larger hot spot containing all of the necessary functions, which may be achieved through hot spot detection parameter tuning. Lastly, hot spot 0 contains initial start up routines along with routines used commonly throughout the application. Control often transfers back to this hot spot for execution of these routines along less frequent paths. In particular, parts of the marking function are shared between hot spots 0 and 3. Again, with further tuning, even cleaner separation of these hot spots might be possible. It is important to note that since trace generation and optimization are performed on a hot spot basis, for *130.li* the mechanisms need only be invoked six times. In addition, the mechanisms not used after the first 6.5M instructions of execution.

5.8.1 Control-flow benefits of extracted traces

Figure 5.9 summarizes the reduction in taken control-transferring instructions due to the code straightening and fetch optimization. Each pair of bars for a benchmark is normalized to 100% of the taken control transfers in the original code. The bars for the optimized applications include taken control transfers from both the code cache and the original code. On average, a 45% reduction is seen across the benchmarks, verifying the effectiveness of the code-straightening techniques. Notice that call and return inlining is particularly effective, removing 16% of the taken control transfers in *130.li*, and sizeable amounts in the other benchmarks. Code straightening techniques for the conditional branches yield an average 24% reduction

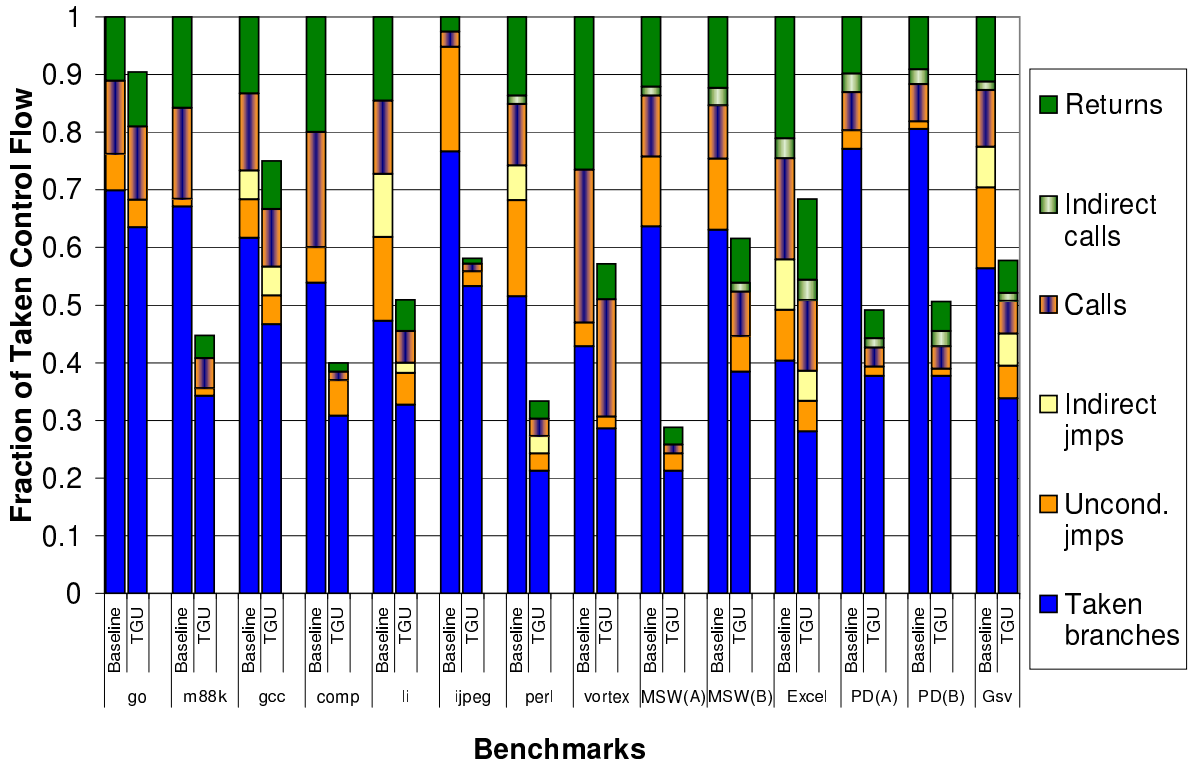


Figure 5.9 Reduction in taken control-flow instructions in optimized code compared to original code.

in taken control transfers, and as much as 40% for *MSWord(A)*, *PhotoDeluxe(A)*, and *PhotoDeluxe(B)*. These results show a dramatic reduction in the number of taken branches.

5.8.2 Processor fetch performance of extracted traces

Table 5.5 presents the results of the hot spot detection and trace generation system with fetch optimizations. A large percentage of dynamic execution occurs in instructions from the code cache. Typically less than one percent of instructions are executed while the TGU is looking for traces to form within a hot spot (Scan/Pending Modes), and a very small percentage of instructions are executed while actually writing the instructions to the code cache (Fill Mode),

Table 5.5 Benchmark detection and trace generation results.

Benchmark	% Insts. from Code Cache	% Scan/ Pending	% Fill Mode	Code Size(KB)	Entry Points
099.go	10.51	1.02	0.0051	14.8	60
124.m88ksim	68.91	4.98	0.0027	8.6	49
126.gcc	32.01	1.07	0.0063	135.1	715
129.compress	87.05	0.84	0.0001	6.4	30
130.li	74.32	0.61	0.0032	13.6	59
132.ijpeg	84.44	0.09	0.0005	22.2	57
134.perl	72.34	0.04	0.0002	12.4	69
147.vortex	34.08	0.12	0.0006	26.4	103
MSWord(A)	78.46	0.08	0.0014	10.2	37
MSWord(B)	45.66	0.29	0.0040	73.9	330
MSExcel	30.69	3.12	0.0271	87.6	352
PD(A)	86.38	0.58	0.0030	18.9	105
PD(B)	81.15	1.25	0.0107	19.1	101
Gsview	60.15	0.35	0.0027	61.0	336
Average	60.44	1.03	0.0048	36.4	172

often less than 0.005%. Even if the writing process requires a several cycles per code cache instruction, the total overhead would be well under 0.1%.

To evaluate the effectiveness of the layout optimizations, each benchmark was simulated with several different fetch unit configurations. Figure 5.10 shows the performance of the various fetch mechanisms. As the optimizations were targeted toward the fetch unit, the *Fetched Instructions Per Cycle* (FIPC) metric was selected as an appropriate gauge of effectiveness. The bars of the graph compare the FIPC of various processor models to a baseline configuration of an aggressive multiple-block fetch unit operating on the original code. The first bar depicts the FIPC for a processor with hot spot detection and trace generation hardware, which averages 22% improvement over the base case. The improvement achieved by a comparably sized trace cache is 18%. Adding the trace cache in addition to the proposed hardware yields

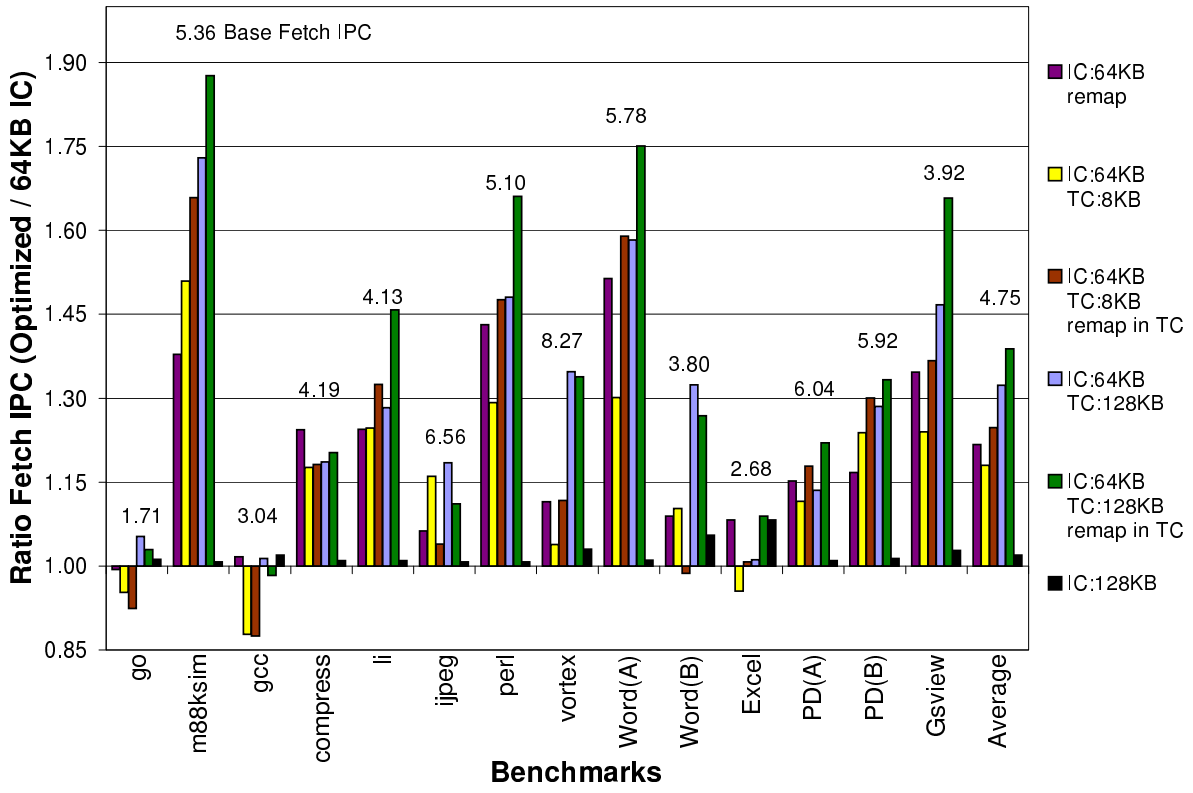


Figure 5.10 Fetched IPC for various fetch mechanisms.

a benefit of 25% over the base case. With a much larger trace cache, approximately 15 times larger in size than the hot spot hardware, the FIPC is improved to 32% over base, and 39% if hot spot hardware is also included. Doubling the size of the traditional instruction cache has a significantly higher hardware cost than the hot spot hardware, and realizes only a 1% improvement. Despite the large reduction in taken control transfers shown in Figure 5.9, FIPC does not necessarily scale accordingly. This is primarily because branch mispredictions cause a dramatic number of stall cycles, which lessens the effect of improving throughput during useful cycles.

One advantage of the TGU system over a trace cache is its ability to inline returns and indirect branches as was seen in Figure 5.7. Our traces may also include loops, as was shown in Figure 5.3. Conveying loop structure potentially allows better optimization than could be performed with simpler traces.

CHAPTER 6

REGION OPTIMIZATION

The goal of Explicitly Parallel Instruction Computing (EPIC) [10] is to increase application performance by enabling higher levels of instruction-level parallelism while maintaining reasonable hardware complexity. In EPIC processors, much of the complexity associated with dynamic scheduling performed in a traditional out-of-order execution core is moved from run time to compile time, as the onus is placed on the compiler to determine which instructions can be executed in parallel. The assumption is made that through global analysis and meticulous scheduling the compiler can achieve an efficient specification of instruction execution, known as a plan of execution (POE) [10]. This plan details when, where, and with what resources operations should execute. EPIC instruction set architectures are designed to facilitate the communication of this POE to the processing core.

In order to profitably optimize a program, state-of-the art compilers utilize profile information to determine the frequencies of various paths of execution through the program. This information allows the compiler to make transformations that are statistically the most beneficial to the performance of the program. Because the compiler chooses a static POE, however, EPIC processors are less able to exploit factors which can only be determined at run time compared to out-of-order processors that engineer their POE at run time. During execution, the common paths through the program may differ from those taken during compile-time profiling. Even

when the compiler-assumed profile is accurate, some paths of execution may be only important during certain phases of program execution and insignificant during others. Some of these important run-time execution paths may cross software boundaries such as dynamically linked library interfaces. A static compiler would not have simultaneous access to code from both sides of such an interface for effective optimization. Furthermore, a compile-time generated POE is also less able to account for dynamic variations in the underlying microarchitecture, such as load latency due to cache misses. Unfortunately, many of the very features of EPIC that empower the compiler also significantly complicate an out-of-order pipeline design [29].

Run-time Optimization Architecture (ROAR) is an extension of EPIC principles that will allow adaptation and optimization of the static POE at run time when transformations become desirable, thereby exploiting dynamic program behavior. While ROAR has been designed as an extension to EPIC architectures, future superscalar architectures could utilize ROAR techniques to potentially eliminate the need for the constant dynamic scheduling of out-of-order processors. The ROAR extensions consist of two components: the instruction Scheduler and Optimizer, jointly named SHOP. The design of the instruction scheduler is detailed in this work while the optimizer is assumed to be a microcoded engine. The scheduler is a hardware component that performs cycle-by-cycle scheduling from a window of instructions (much like a traditional out-of-order scheduler [19]) but is structured more like a software scheduler that tracks all of the interinstruction dependences.

The TGU forms sequences of basic blocks into larger single-entry multiple-exit regions that are similar to superblocks. In order to avoid side entrances into traces, the TGU either performs tail duplication in the form of a new trace off of a side exit, or inserts a transition back

to original code where execution would remain until the entry-point is once again encountered. Because the traces are often formed from several original blocks (even original superblocks), additional superblock optimization [30] can be beneficial.

6.1 Rationale Behind Hardware-Based Optimization

Several potential approaches to dynamic optimization have been proposed that range from pure software systems that run on the same processor as the target application, to hardware-only optimizers in the spirit of out-of-order execution. Clearly, many of the steps in the run-time detection and optimization process require a significant amount of time and memory. The goal is to minimize the overheads which detract from performance increases gained through optimization. We would like to ensure maximal performance through full-speed execution during opportunity detection and subsequent optimized execution while limiting any slow-downs during optimization.

Since we propose only infrequent scheduling and optimization, an effective software optimizer may be possible. However, to be truly effective, optimizations must be performed as close the beginning of a new phase of execution as possible. The speed of optimization performed in hardware allows such systems to benefit from executing optimized code during a greater portion of program execution. In addition, the rapid nature of a hardware scheduler allows it to improve the performance of program phases that are too short to be lucrative for a software scheduler. Software optimizers are also likely to require operating system support, while it is the goal of this work to provide optimization that is transparent to software.

While out-of-order mechanisms are able to dynamically correct for execution anomalies, such as cache misses, they are also *on-line* or on the critical path of the processor, and are thus continuously operating and consuming power. However, by relocating the code restructuring and rescheduling mechanism to the back-end of the processor, essentially taking this hardware *off-line*, many of the benefits of out-of-order execution can be achieved without the latency and complexity of altering the critical path of the pipeline itself. With this mechanism, rescheduling is performed only on a need basis thus eliminating the constant energy consumption. While the current evaluation of ROAR does not necessarily prove occasional rescheduling to be the best design, it does show initial evidence and provides a framework for future study.

6.2 Compilation Support

During static compilation, compilers can perform aggressive pointer aliasing analysis [31] to permit safe reordering of some loads and stores. Memory dependence arcs could be drawn between loads and stores in different function bodies, as shown in Figure 6.1(a). However, this program-scoped analysis is generally condensed into intrafunction memory dependence arcs for use by the compiler's scheduler, as shown in Figure 6.1(b). Because a compile-time scheduler typically does not move loads and stores between function bodies, memory dependences to instructions in called functions are simply represented by arcs to the call instruction itself. Although valuable to a run-time optimizer, all of this dependence information is typically lost once generation of the executable is complete.

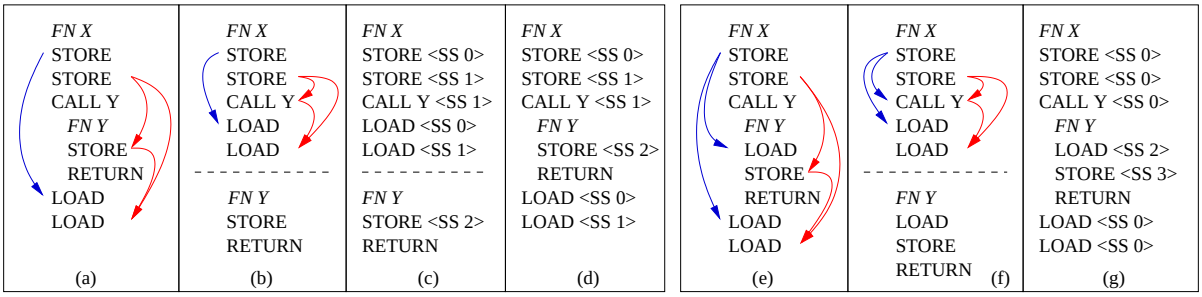


Figure 6.1 Memory dependences. (a) Global memory dependences. (b) Reduction to intra-function dependences for compiler scheduling. (c) Intrafunction store set allocation. (d) Trace scheduling of memory operations. (e)-(g) Dependence derivation for a callee which accesses multiple memory dependence chains.

To compactly communicate this information to the run-time optimizer, we modified the compiler to further reduce the memory dependence information into a single per-instruction identifier, called a *store set*, similar to what is defined in [32]. Instructions that potentially read from or write to the same memory locations are dependent and will have the same identifier, as shown in Figures 6.1(c) and (d). Like the dependence arcs, the store sets also have intrafunction scope. However, because a store set on a call implies that an instruction in the callee could read or write the memory location associated with it, the relationship between calls and loads on the same store set is ambiguous and conservative assumptions must be made. Furthermore, allowing a single store set identifier per operation may limit the dependence resolution achieved through the store sets. Consider the call instruction in Figures 6.1(e)-(g), where a single store set on the call must represent the read of one memory chain and write of another. This situation leads to a loss of resolution in that the two memory dependence chains have effectively been merged.

Store set allocation begins by grouping together instructions that could potentially reference the same memory objects. The groups of instructions are formed from the memory dependence arcs stored on each instruction in the compiler's intermediate representation. Even though a single instruction may reference several different independent memory objects, it can only have a single store set identifier, like the previously described situation with call instructions. For example, a load instruction C may read from memory locations stored to by either A or B depending on the pointer used in instruction C. Even though the two stores may be to distinct memory locations, they will have the same store set because C must have the same set as A and C must have the same set as B. Once the groups of instructions are determined, the groups are allocated in a round-robin fashion to the available store set identifiers. Distinct stack references, such as spills and fills, are allocated to a special stack store set, which is the highest number identifier in the current implementation. The stack store set indicates that simple operand analysis can uniquely identify the specific memory location and that the memory location cannot be pointed to by any generic pointer. This feature allows a dynamic code optimizer to perform optimizations such as store-to-load forwarding because the memory locations can be perfectly disambiguated. Like spills and fills, parameters passed on the stack can also be allocated to this special identifier provided that the address of the parameter is never taken and stored as a pointer. Use and implementation of store sets is further discussed in Subsection 6.4.6.

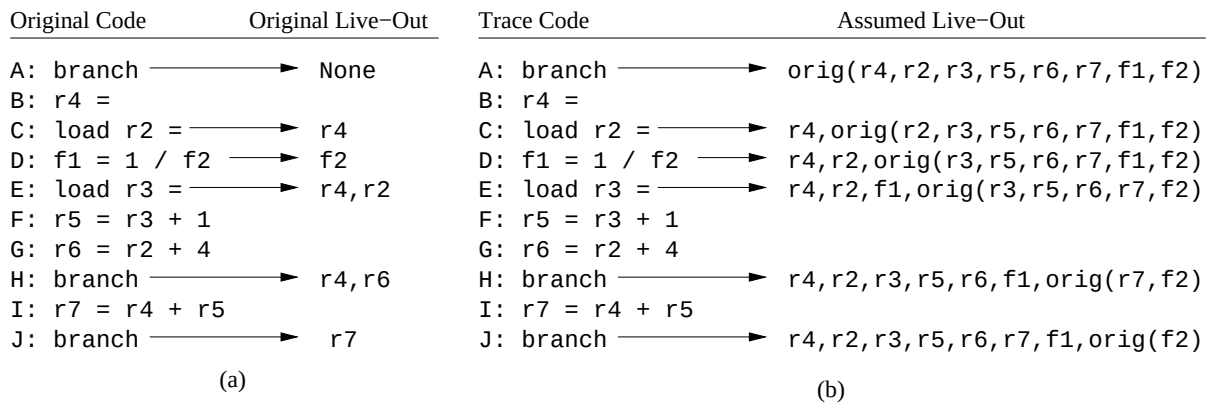


Figure 6.2 Live-out register analysis results for a sample code segment. (a) Aggressive, global analysis performed by the compiler. (b) Local (trace) analysis.

6.3 Instruction Scheduling Support

Some programs utilize exception handling as a mechanism to recover from errors or as a means for structured control flow. These applications often rely on the notion of *precise exception handling* which dictates two properties: first, that the exceptions occur in program order (the *ordering* property), and second, that expected program state is consistent with program order when an exception occurs (the *liveness* property). When the compiler reorders instructions, it has the global analysis capability determine which registers and memory locations need to be available into exception handlers and which do not.

Consider the code sequence in Figure 6.2(a). The branches and potentially excepting instructions (PEIs) each have a control-flow arc that is annotated with the registers that are live along the path. Even though the load instruction `C` and the floating-point divide instruction `D` have no data dependences between them, the programmer may insist that any exceptions

caused by these two instructions occur in the specified order. Both the compiler and any post-link optimizer must preserve this ordering.

The compiler and post-link optimizer must also preserve the liveness of registers into exception handlers and along branch paths. For example, add instruction `I` writes `r7` which is not used along the taken path out of the previous branch instruction `H`, so `I` could be safely speculated above `H`. However, at runtime, without global analysis, it is not known whether the original value of `r7` is live along the taken path from `H` and therefore `I` must be pinned below `H`. The conservative liveness requirements are shown on the arcs in Figure 6.2(b).

6.3.1 Conceptual description of Precise Speculation

To enable run-time code reordering while preserving precise exceptions without global analysis, *Precise Speculation* [11] was developed as an enhancement to Sentinel Speculation [33]. Precise Speculation allows for speculation of individual instructions while eliminating the need for explicit recovery code generation. Consider the conceptual example in Figure 6.3. The right column of circles represents a sequence of instructions from the original code in program order from top to bottom. Suppose that the run-time instruction scheduler wishes to move instructions `A` through `C` higher in the schedule, above load instruction `X`. Simply moving them to their new locations would violate the precise exception requirement by overwriting the destination registers of `A` through `C` even though the original values in those registers may be needed along the exception path from load instruction `X`. The Precise Speculation technique allows for the reordering, shown in the optimized trace in the left-hand column. When `A` through `C` are moved up in the trace, they are marked as speculative and will write into

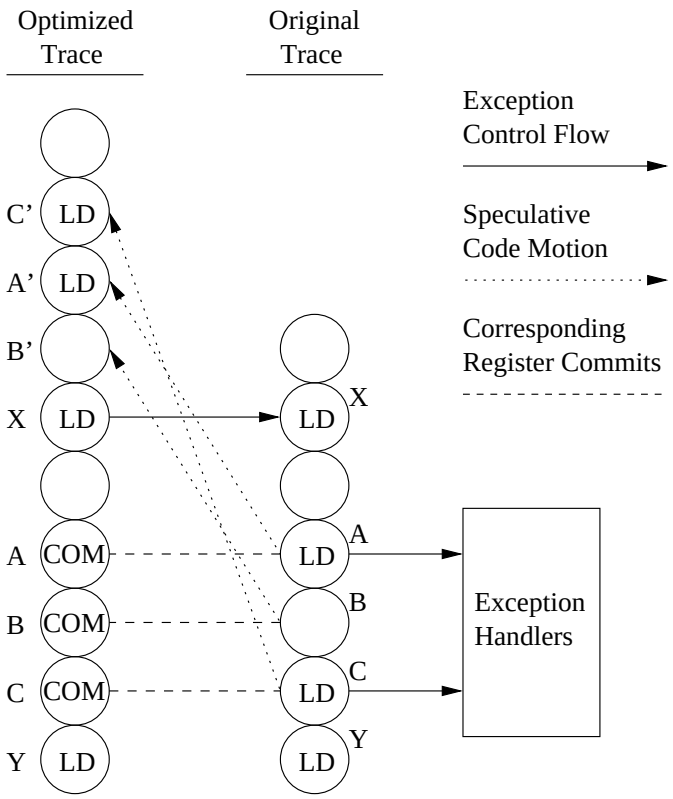


Figure 6.3 Conceptual organization of Precise Speculation.

a speculative register file. A *commit* instruction will be used to copy these speculative registers into their architectural register counterparts inside their original home blocks, as represented by the `COM` instructions in the optimized trace. This ensures the liveness requirement by writing the architectural registers in the appropriate home blocks. Precise Speculation, hence, allows for upward code motion through speculative operations and their corresponding commits, but does not provide for downward motion.

The exception ordering requirement is ensured through a unique recovery code mechanism. In this optimize code sequence, load instructions `A` and `C` have been reordered, potentially violating the requirement. However, whenever a speculative instruction signals an exception,

its signal is suppressed, but a global exception bit is set. Upon the next nonspeculative PEI or branch, all noncommitted values in the speculative register file will be discarded and control will be transfered back to the original code sequence where execution will resume in original order. In other words, if C' were to signal an exception, control will transfer back to the original code through X where A will be allowed to signal first.

6.3.2 Precise Speculation example

Figure 6.4(a) provides an example code sequence that will demonstrate Precise Speculation functionality. To its right, in Figure 6.4(b), instructions 2 and 3 are speculated above branch instruction 1, and 5 is speculated above 4. As previously described, these instructions are marked as speculative and write into the speculative register file in order to preserve the original values that are live along the taken path of branch instructions 1 and 4. All instructions read from the speculative version of the register. Speculative instructions write to the speculative version while nonspeculative instructions write to both. The speculative instructions require a corresponding commit instruction later in the schedule to update the nonspeculative version. This commit occurs in a location control equivalent to the original location of the speculative instruction (COM 7 for load instruction 2, in Figure 6.4(b)). If outstanding, uncommitted register updates exist at a taken branch, those registers' speculative values are restored to their nonspeculative values. The overall model is similar in concept to the speculative register state within an out-of-order processor [34], except that it is visible through an ISA extension to the optimizer. Out-of-order processors write speculative results into reorder buffer entries.

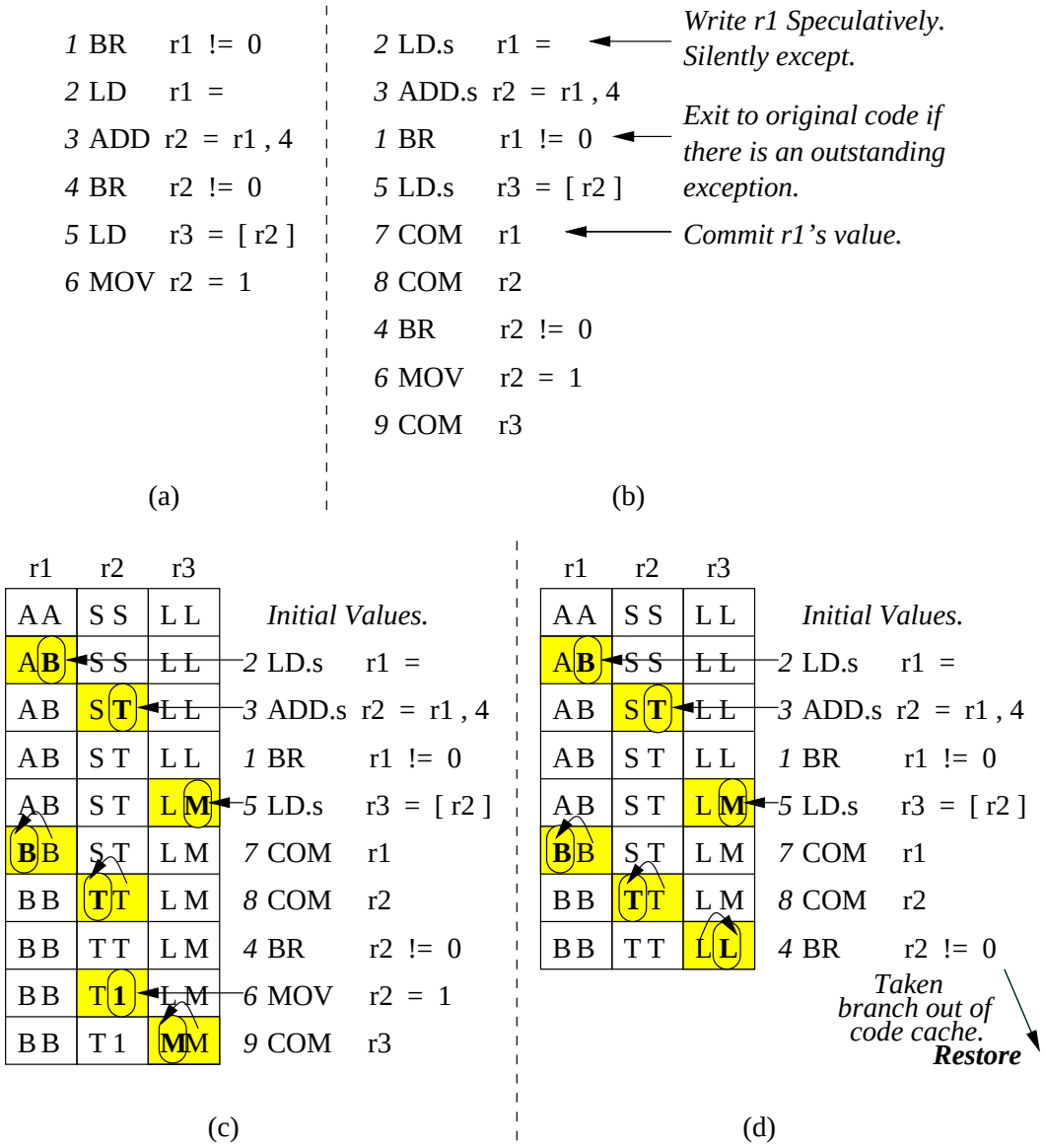


Figure 6.4 Precise Speculation example. (a) Example code sequence. (b) Sequence after speculation of 2, 3, and 5. (c) Execution with no taken branches. (d) Execution when branch 4 takes.

Speculative results are committed to the architectural register file after all prior instructions according to original program order have committed their results.

The states of the speculative and nonspeculative register files are depicted in Figure 6.4(c)-(d) for an execution path that proceeds completely through a trace and for a path that exits the trace due to taken control flow. The column under each register contains the state of the register after the corresponding instruction executes. The value on the left is the nonspeculative value and the value on the right is speculative. To assist in understanding the figure, times when a register value changes are highlighted and the value that changed is in bold and is circled. The letters in the columns represent arbitrary values. Note how load `2` changes `r1`'s speculative value to `B` while preserving the nonspeculative value. At commit `7`, `r1`'s speculative value is preserved by copying it to the nonspeculative version.

Figure 6.4(d) proceeds in much the same way as Figure 6.4(c) until branch `4` is encountered. In this example, branch `4` does not fall through. Since the branch is taken, all outstanding nonspeculative values are restored to their nonspeculative state. In particular, this restoring action causes r3's speculative value to change from `M` back to `L`. Since the register file can be restored to a nonspeculative state at any nonspeculated instruction, precise exceptions can be supported by simply transferring execution from a rescheduled trace to the equivalent point within original code upon an exception condition. At that point, the original instructions serve as the recovery code which will execute the speculated instructions in their original nonspeculative order to provide the expected environment. This model does, however, require that all speculated instructions be moved up above at least one nonspeculative PEI or branch. This instruction will serve as the exception check that will force a transition back to original code for recovery

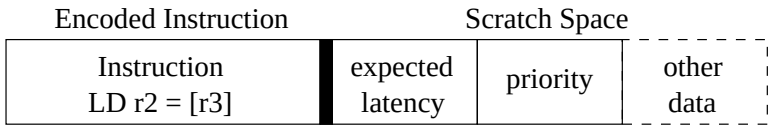


Figure 6.5 Instruction Trace Buffer entry with fields for the instruction and analysis scratch space.

if any of the speculated instructions should have signaled an exception. Commit instructions in this model are referred to as *rolling commits* because they greedily commit state throughout execution of the trace. A more detailed description of the Precise Speculation mechanism and its implementation can be found in [35].

6.4 Instruction Scheduling Architecture

After trace generation for a new hot spot has been completed by the TGU, the individual traces are filled from memory into the Instruction Trace Buffer, shown in Figure 6.5. Once the trace is filled, optimization and analysis can be performed on the trace prior to instruction scheduling.

6.4.1 Analyzer and optimizer

Beyond instruction scheduling, optimizations that alter and transform the instruction sequence may also improve application performance. Trace-based or superblock-based optimizations are prevalent in modern compilers and have also been evaluated in other dynamic optimization frameworks. In DAISY [36], optimizations such as copy propagation, instruction

combining (constant operand reformulation for instructions moved across other instructions that have constant modifications to a common register operand), loop unrolling, load-store telescoping (a specific case of store to load forwarding), tree-height reduction (expression reassociation), and unification (merging of common instructions from opposite sides of hammock-shaped regions) are employed. In addition, rePLay [20] includes common subexpression removal, subroutine inlining, strength reduction, common idiom strength reduction and folding, dead code elimination, and further utilizes zero cycle register moves. DAISY uses tree-shaped regions as its unit of optimization while rePLay uses traces.

Use of precise speculation does somewhat limit the types of optimizations that can be applied, but in exchange it allows the previously described rolling commits. At the bottom exits of the trees and traces in DAISY and rePLay, all registers must contain the same values as they would at the same points in the original code. Side exits in these frameworks force rollbacks to the beginning of the trees and traces with resumption of execution back in original code. Precise Speculation allows each nonspeculated potentially excepting instruction to be a true exit to original code; therefore, the same register matching requirement at the end of trees and traces is present throughout ROAR traces. While the speculative register files in ROAR allow for early computation of values, computations cannot often be pushed much lower in the schedule because of frequent side exit points. Thus, instruction scheduling is limited to upward code motion. Similarly, instructions can rarely be eliminated because of the likelihood of their results being necessary along side exits into the original code. This prevents most dead code elimination and limits optimizations to those in the *forward* direction. Copy propagation is an excellent example. Consider a value in register A that is moved into register B. Backward copy

propagation would eliminate A by changing its producer to generate B directly (and changing all uses of A into B). However, B cannot be written early because the old value in B may be needed along side exit paths up until the move instruction, and A may be needed along those side exits as well. Forward copy propagation leaves the move of A into B in place for correctness along the side exits, but changes all uses of B into A so they can be speculated up above the move.

Other types of analysis can be performed on the trace while it is in the Instruction Trace Buffer. For example, a dependence height analysis is performed bottom-up on the stored trace to compute scheduling priorities for the instructions. During the scheduling process, instructions with higher dependence heights are given priority when multiple instructions are ready to be issued simultaneously. This analysis comprises a single linear pass that computes the dependence height (minimum cycle separation due to dependences) between instructions and eventual end consumers, generally branches or stores. Dependence heights are calculated instruction-by-instruction, where the height of each instruction is the maximum of the heights of its consumers plus its latency. In this scheme, the height values are used as the priorities: the greater the latency chain to eventual consumers, the greater the need to schedule it early in the trace and the greater the priority.

However, a particular instruction's result may not be needed until late in the trace (giving it scheduling freedom), but it may be required earlier for use through some of the side exits. A heavily taken loop back branch may appear as a side exit if the TGU creates a superblock trace by appending the code along the fall-through path of the loop back branch to the end of the loop block. Stated more generally, an instruction with low importance may be delayed in the

schedule in favor of speculated instructions from below that have higher dependence heights. However, by delaying it, all subsequent nonspeculable instructions (branches and stores) are also delayed, because Precise Speculation only allows for upward code motion. In these cases, it is critical to schedule the loop back branch and the instructions it depends on (in effect, all earlier instructions) early in the trace. Each cycle that the loop back branch is delayed adds an additional cycle to each iteration of the loop. To schedule these instructions early, branches with high taken percentages in the BBB are given the dependence height that is one higher than the maximum computed to that point in the trace during the backward pass. Thus, instructions that feed the branch will be moved even higher.

The actual design of the SHOP optimizer component is beyond the scope of this current work and is left for future research. The optimizer would likely consist of a microcoded engine capable of operating on the instructions in the buffer. The microcoded feature would allow for diverse optimizations as well as upgradeability. Once analysis and optimization are complete, the traces are fed into the SHOP scheduler component.

6.4.2 Scheduler Table architecture

The scheduler component of the SHOP, called the Scheduler Table, is designed to perform List Scheduling [37], [38] over the instructions in a trace. It produces a new POE targeted for the specific microarchitecture of the host machine. The table is constructed as a circular queue of entries, each of which contains an instruction and fields used to manage incoming dependences from other instructions. The new POE will be constructed by stepping through

the cycles in the new plan scheduling instructions with satisfied incoming dependences and high priorities.

The Scheduler Table contains a set of instructions enqueued into the table, shown on the left in Figure 6.6, with a partially completed POE on the right. The entries are filled into the table after the Youngest Instruction (queue tail) in original program order to maintain proper data dependences. The dark (red) boxes represent instructions that have yet to be scheduled into the new plan, where circled entries are those for which all dependences have been satisfied (ready). The light (blue) boxes represent those that have been scheduled into the new plan, where crossed entries are those for which their latencies have been satisfied in the plan and their dependent operations have been made ready. In Figure 6.6(a), `I33` and `I34` were scheduled in the current cycle and therefore their dependents cannot yet be freed. `I33` is the Oldest Unfinished Instruction (queue head) in the list and is tracked as such. `I35` has been scheduled in a previous cycle and its latency has been satisfied. However, its entry cannot be freed until it is the oldest instruction in the table so that the original program order of the queued instructions is always maintained. `I36` is the first ready instruction and is tracked as the Oldest Unscheduled Instruction. It will be included in the set of ready instructions, those with satisfied incoming dependences, that will be considered for scheduling into the current slot of the current cycle. The Oldest Unscheduled Instruction also represents the current basic block being scheduled; younger instructions may be ready but may have to be made speculative to be scheduled in the current cycle and slot. Figure 6.6(b) depicts the scheduler state immediately after all ready instructions have been scheduled. In Figure 6.6(c) scheduling has moved to cycle *c+1*. All scheduled instructions whose latencies are now satisfied—`I33`, `I34`,

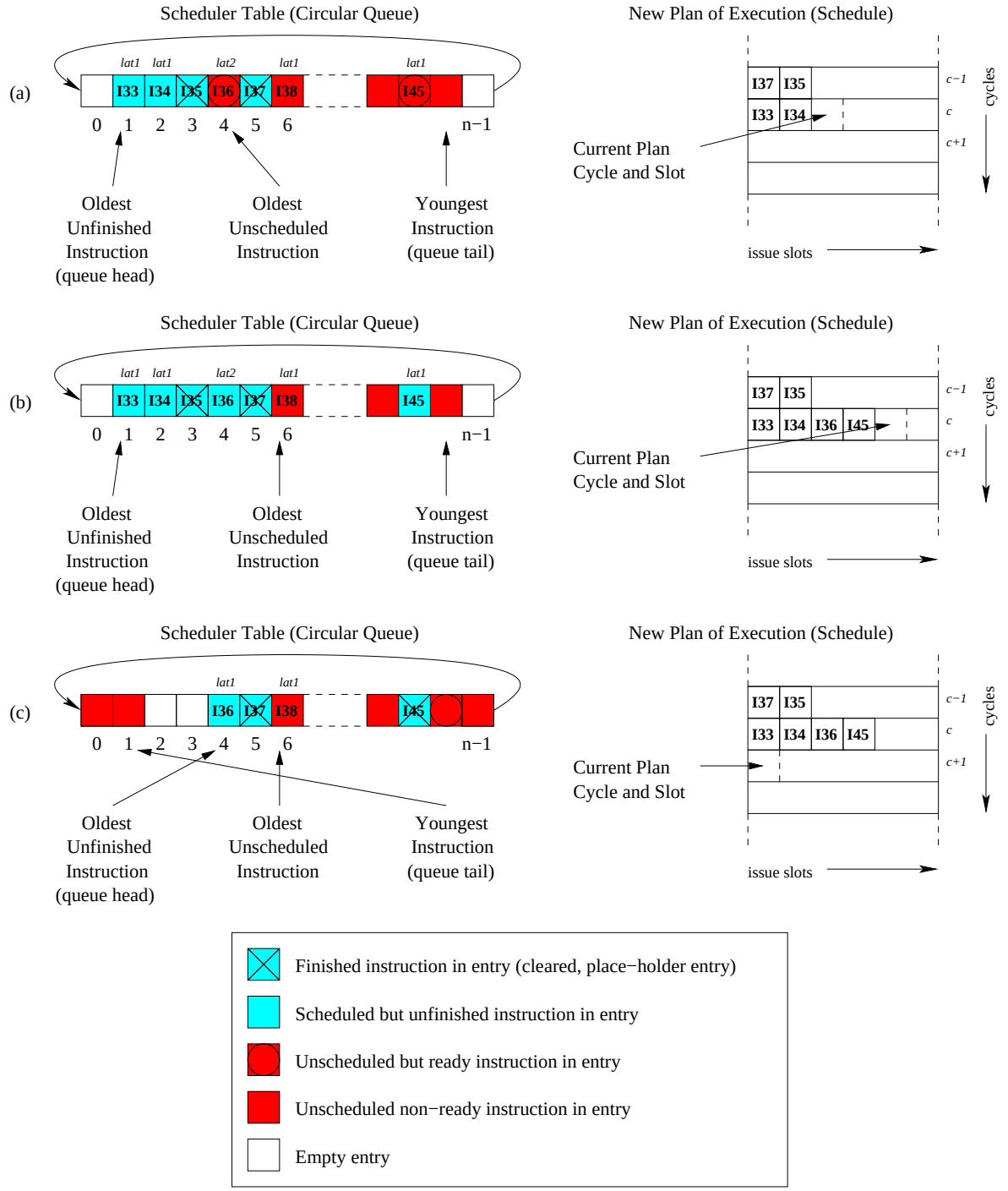


Figure 6.6 Scheduler Table overview. (a) Scheduling midway through cycle *c*. (b) Scheduling complete after `I36` and `I45` because there are no other ready instructions. (c) Scheduling advances to cycle *c+1* leaving `I36` as the Oldest Unfinished Instruction since its latency is not satisfied at this cycle in the schedule.

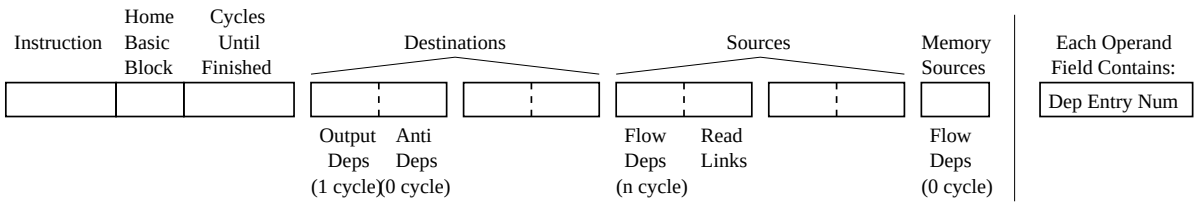


Figure 6.7 Hardware Scheduler Table entry.

and `I45`—are marked as finished. Finished instructions at the head of the queue, `I33` and `I34`, can be freed and used to fill new instructions after the tail.

The scheduling table consists of a set of entries as detailed in Figure 6.7. Each entry contains the instruction being scheduled along with dependence fields for each operand. Each operand field contains the table entry number for the instruction upon which the operand is dependent. Memory instructions also create dependences between themselves which are tracked in memory operand fields in the entry. A discussion of memory dependence tracking is reserved for Subsection 6.4.6.

Figure 6.8(a) diagrams the three register dependence arc types along with the register read arc chain in the form that they would be filled into the table entries. The read chain links all of the readers of a particular register together and will be used to repair antidependence arcs when the readers are scheduled in a different order. For example, the subsequent writer (`A` in Figure 6.8(a)) of a register may be antidependent on the last read of the register's previous version (`B`). However, `A` must be made antidependent on the second-to-last read (`C`) if the last read is scheduled before the other reads. A more detailed exampled is presented in Subsection 6.4.5.

Two other tables are also utilized to track the last write and the last read of each register for the purposes of quick location of the source indices for all dependence arcs. While control

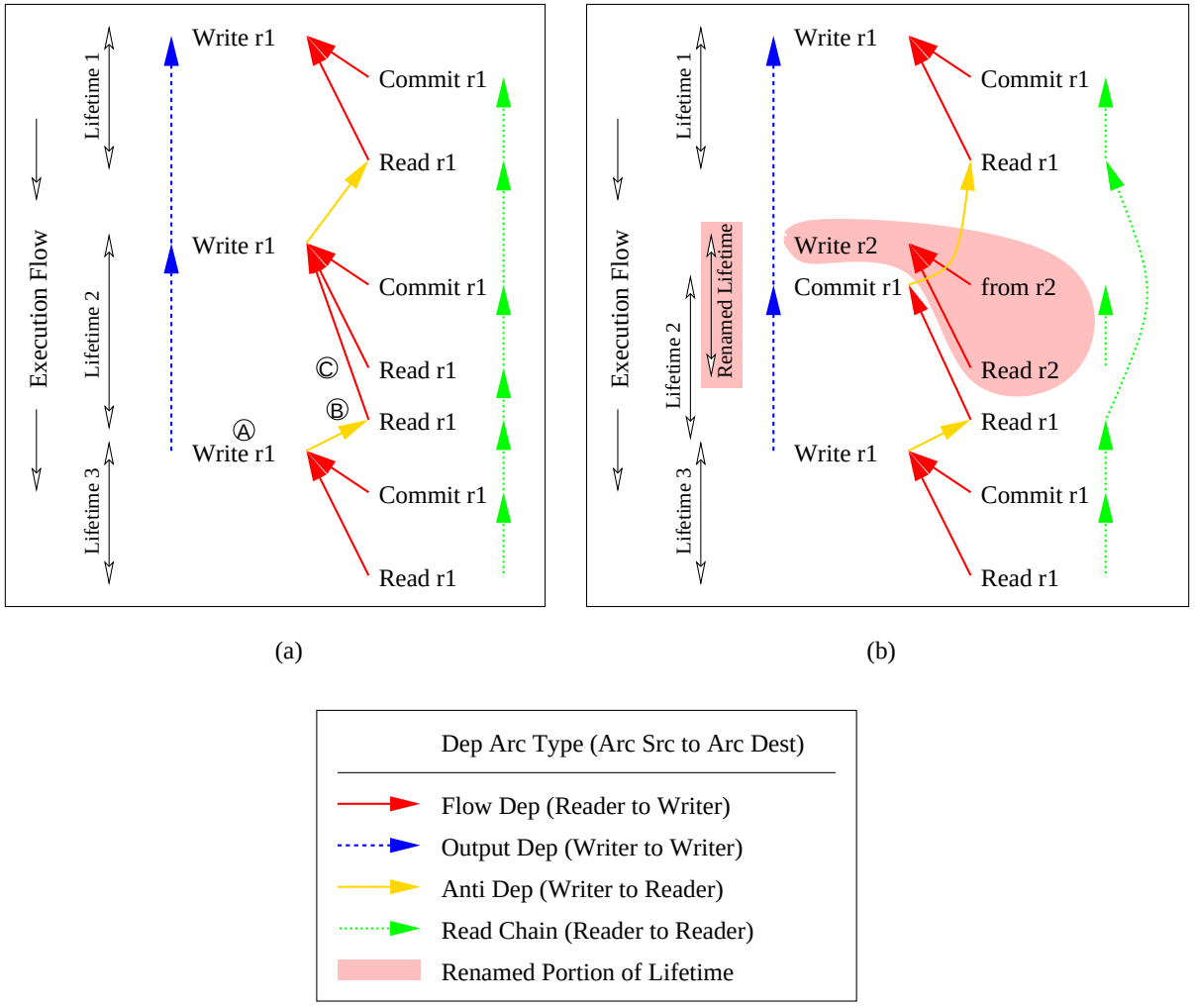


Figure 6.8 Diagram of dependence arc types (arcs always point to older instructions). (a) Standard arc configuration. (b) Arc configuration after renaming a portion of the second `r1` lifetime.

dependences may be relaxed when using the Precise Speculation model previously described, data-flow dependences must be obeyed to ensure correct computation flow. Anti- and output dependences must also be obeyed, unless explicitly renamed (Subsection 6.4.4), so that the values stored in registers are not prematurely overwritten.

```
Main_Scheduling_Loop() {
 0:  While any instructions in the trace remain unscheduled {
 1:      Enqueue_Instructions_Into_Available_Table_Entries();
 2:      Schedule_Plan_of_Execution_Cycle();
 3:      Advance_to_Next_Plan_Cycle_and_Free_Finished_Instructions_From_Table();
}
```

Figure 6.9 Scheduler algorithm: Main driver loop.

6.4.3 Scheduler Table algorithm

The scheduling algorithm in Figure 6.9 consists of three main steps that iterate while instructions in the trace remain unscheduled. An iteration is completed for each cycle in the new POE as instructions are scheduled cycle-by-cycle. First, instructions are filled into the Scheduler Table queue and incoming dependences for those instructions are constructed. Second, instructions with satisfied incoming dependences (ready) and high priorities are placed into the schedule for the current plan cycle. Last, out-going dependences from scheduled instructions are freed, creating a new set of ready instructions, and the current plan cycle is advanced to the next cycle.

In more detail, Figure 6.10 describes the necessary steps to enqueue an instruction into the Scheduler Table. For each instruction inserted into the table, its source and destination register numbers are used to index into the Last Read and Last Write Tables to locate the most recent reader and writer of the particular operands. For example, the index of the most recent writer of a current source register is used to construct a flow dependence by tagging the current source register field with the writer's index. If the Last Read or Write Table entry for the register

```
Enqueue_Instructions_Into_Available_Table_Entries() {
 0: For each available entry after Youngest (queue tail) And
        before Oldest Unfinished (queue head) {
 1:     Enqueue instruction into entry after tail and set Youngest (queue tail) to entry;
 2:     Read index in Last Write Table for each source and
            set flow dependence indices in entry;
            /* Note that dependences always point to older instructions */
 3:     Read index in Last Write Table for each destination and
            set output dependence indices in entry;
 4:     Read index in Last Read Table for each source and
            set read chain indices in entry;
 5:     Read index in Last Read Table for each destination and
            set antidependence indices in entry;
 6:     Update Last Write Table with current index for each destination;
 7:     Update Last Read Table with current index for each source;
 8:     Assign entry the current home block counter value;
 9:     If (instruction is control altering) Then
10:         Increment home block counter;
11:     If (instruction is speculable) Then {
12:         Enqueue a potential commit instruction for speculable instruction
                destinations in next entry;
13:         Set flow dependence index of commit to index of speculable instruction;
14:         Update Last Read Table with commit index for speculable
                instruction destinations;
15:     }
16: }
}
```

Figure 6.10 Scheduler algorithm: Enqueue instructions.

is empty, then there is no previous reader or writer and the dependence field in the entry is left empty. After the dependences are constructed for the current instruction, its source and destination registers are set as the most recent readers and writers in the Last Read and Write Tables. The entry is then tagged with a home basic block identifier that is incremented each time a control-altering instruction is enqueued. This identifier will be used to determine which

instructions are speculative during the scheduling step. Last, if the enqueued instruction is a type that could be speculated, then a potential commit instruction is enqueued in the table immediately following the speculable instruction and is made flow dependent on the speculable instruction. The commit instruction will only be included in the plan if the potentially speculable instruction is actually speculated. If included, it can only be scheduled inside its home block so that the result of the speculated instruction is written in original program order.

The cycle scheduling step is described in Figure 6.11. In this step, each issue slot in the current cycle is considered one-by-one. For each issue slot, the ready instructions are gathered and considered for scheduling into the slot. An instruction is deemed ready if all of its incoming dependence arcs are satisfied, if its resources are available, and if it is being scheduled into an acceptable home block. Its incoming dependence arcs are recognized as satisfied if the operand fields in the scheduler entry are empty. Operand fields can be empty if there was no producer in the Last Read or Write Tables when the instruction was enqueued, depending on dependence type, or if the producing instruction has been scheduled or finished, also depending on type. Instruction scheduling and finishing will be discussed shortly. Of the instructions with ready operands, the ones with highest priority are explored first. An instruction is scheduled if it meets two criteria; otherwise, the next highest priority instruction will be explored. First, the resources required by the instruction, for example, functional unit, must be available. Second, the instruction must be scheduled into an appropriate home block. As previously mentioned, the Oldest Unscheduled Instruction represents the current home block being scheduled. From this instruction, all older instructions have been scheduled along with some speculative younger instructions. Branches and stores cannot be speculated under the Precise Speculation

```
Schedule_Plan_of_Execution_Cycle() {
 0: For each issue slot in cycle {
 1:     For each ready instruction in priority order {
 2:         If ( (resources for instruction are not available) Or
                  (instruction is a store or branch And it is not the Oldest Unscheduled) Or
                  (instruction is a speculable PEI And
                       currently in its home block And
                       it is not the Oldest Unscheduled) Or
                  (instruction is a commit And currently not in its home block) ) Then
 3:             Continue with next instruction;
 4:         Insert instruction into Plan of Execution in current cycle at current slot;
            /* Instruction is now scheduled but not finished */
 5:         Broadcast free-or-update-antidependence message;
 6:         Broadcast free-or-update-read-chain message;
 7:         Broadcast free-memory-dependence message;
 8:         Free sources from Last Read Table;
 9:         Set finished field with instruction latency;
10:         If (speculated) Then
11:             Broadcast update-home-block-message;
12:         Else
13:             Free potential commit in next entry;
14:         If (instruction is Oldest Unscheduled) Then
15:             Move Oldest Unscheduled pointer to next oldest but not scheduled;
16:     }
17:     If (unable to schedule any instruction in slot) Then
18:         Break scheduling for current cycle;
19: }
}
```

Figure 6.11 Scheduler algorithm: Schedule instructions.

model and therefore must be scheduled in their home blocks. Since these instructions represent the end of their blocks, all older instructions must be scheduled prior to these so that older instructions are not moved downward by being scheduled later. Therefore, they must be the Oldest Unscheduled Instruction before they can be scheduled. Speculable instructions, under Precise Speculation, must either be scheduled in their own home block or across a

nonspeculative instruction that will serve as its check. Therefore, speculable instructions can either be scheduled in their own home block, and therefore must be the Oldest Unscheduled Instruction like stores and branches, or must have a home block greater than that of the Oldest Unscheduled Instruction. Last, commit instructions can be scheduled any time in which their home block matches that of the Oldest Unscheduled Instruction.

When an instruction is scheduled, it is placed into the resulting code in its proper location. At this time, all antidependences from the source operands in the scheduled instruction to subsequent writes can be freed because a write can follow a read of the same register in the same cycle. As previously described, antidependences must be updated to the second-to-last read if the last read is being scheduled. Therefore, a message is sent to all other entries specifying that antidependence fields with the current instruction's index should be updated to the index stored in the current instruction's read chain field. If the current instruction has an empty read chain field, then the antidependence field can be emptied. Likewise, the read chain is also updated so that all read fields with the current instruction's index are also updated to the index stored in the current instruction's read chain field. The antidependence and read chain fields may need to be annotated with the register number in order to adjust the links for the correct source operand. Now that the current instruction has been scheduled, it can be cleared from all operand entries in the Last Read Table; newly enqueued instructions will not require antidependence or read chain arcs since the instructions can no longer be reorder.

At this point, the finish time entry field is set to the instruction's latency, which will be used to free flow dependences later. If the current instruction was speculated out of its home block, it will no longer serve as an exception check and also no longer serves to end a home

```
Advance_to_Next_Plan_Cycle_and_Free_Finished_Instructions_From_Table() {
 0:  For each scheduled instruction in table {
 1:      Broadcast free-output-dependence message;
 2:      Decrement instruction finish time by 1 cycle;
 3:      If (instruction has finish time of 0 cycles) Then {
             /* Instruction has now finished */
 4:          Broadcast free-flow-dependence message;
 5:          Free destinations from Last Write Table;
 6:          Clear entry from table;
 7:          If (instruction is Oldest Unfinished) Then
 8:              Move Oldest Unfinished pointer to next oldest unfinished;
 9:      }
10:  }
}
```

Figure 6.12 Scheduler algorithm: Advance to next cycle.

block. Therefore, its home block must be merged with the block after it in the trace, so that subsequent instructions will not be speculated using the current instruction as their check. If not speculated, the potential commit that was enqueued after it can be eliminated. Last, the Oldest Unscheduled Instruction pointer is advanced accordingly.

Once scheduling for a particular cycle is complete, preparations for the next cycle begin, as described in Figure 6.12. Output dependences can be freed by emptying output fields that contain the index for any of the instructions scheduled in the current cycle. Then, finish times for all scheduled instructions are decremented by one to account for the schedule moving on to the next cycle. All instructions with finish times of zero will have their destination operands ready for consumption by other instructions in this next cycle and all flow-dependent operand fields can be freed. Instances of a destination operand from any finished instruction can be cleared from the Last Write Table at this time; newly enqueued instructions will not require

flow or output dependence arcs since the instructions can no longer be reordered. Lastly, the Oldest Unfinished Instruction pointer can be updated accordingly, and all entries up to this pointer can be freed for reuse during the next enqueue step.

The algorithm can be shown to produce a valid schedule by examining the restrictions placed on instruction scheduling. First, an instruction can never be placed into the POE until its source operands are ready. Second, an instruction can never write its result early, thus destroying another result, unless it is written to the speculative portion of the register file. Last, either the instruction or the commit of its result must appear in the instruction's original home block, thus yielding the appearance of sequential execution.

6.4.4 Explicit register renaming

Lastly, the Precise Speculation model also enables explicit register renaming. In Figure 6.8(a), three lifetimes of `r1` are shown. The write that produces lifetime 2 is trapped by an antidependence on the last read of lifetime 1. In order to speculate the write for lifetime 2 higher in the instruction schedule, its destination must be modified to reflect a temporary destination, thus freeing the write from the antidependence. Under the Precise Speculation model, the speculative versions of unused registers may be used to hold these temporary values. However, unlike standard commits that simply copy the speculative value of a register to its nonspeculative version, *move-commit* instructions are required to transfer the value from the speculative version of the temporary register to the nonspeculative version of its home register. Even if all subsequent uses of the home register are transformed to use the temporary register, the move-commit must still take place in order to preserve precise exception handling.

In the example in Figure 6.8(b), a portion of the second lifetime with the shaded background was transformed to use `r2`. Note how the move-commit (denoted in the figure as `Commit r1 from r2`) takes on the properties of the original write (output and antidependences). The renamed write and the first subsequent read no longer have any dependences preventing their issue and can be speculated to earlier in the schedule, thus demonstrating the primary benefit of register renaming. The second read in this example was not renamed and is not required to be renamed. It is, however, pinned below the move-commit which actually writes `r1` and also blocks the third write of `r1` through an antidependence. This situation, where only a portion of a lifetime is renamed, may be desirable but could also be an artifact of an incomplete renaming solution. A fraction of a lifetime may be renamed when only a fraction is present in the Scheduler Table when the decision to rename is made. Renaming must continue as new reads for the renamed lifetime are filled into the table in subsequent enqueuing steps. If the second read instruction would also have been renamed, the third writer would only be output dependent on the move-commit, thus enabling nonrenamed speculation of the third lifetime to a point just after the commit. Increased freedom of motion for lifetimes subsequent to the renamed lifetime is the second benefit of renaming. Of course, the third lifetime could also be renamed to another temporary register.

In the Precise Speculation model, values are always read from the speculative version of the registers. However, using a speculative portion of a register to hold a renamed value also disallows normal access to that register. Therefore, temporary registers must be chosen such that they can be restored to their correct nonspeculative values prior to their use. In this work, only registers not touched in the particular trace are eligible to be temporary registers. To

ensure correct values in all registers, restore operations must be performed at all trace exits. The Precise Speculation model ensures restoration on all taken trace exits, eliminating speculated values from deeper in the trace, and naturally clearing renamed values. In addition, an explicit restore is needed at the fall-through exit of the trace to clear the renamed values.

Unlike the standard instruction selection process without renaming, an instruction can be selected for scheduling in a POE cycle if just its register flow and memory flow dependences have been satisfied. Its output and antidependences can be ignored because they will be eliminated when renaming is performed. Once an instruction is chosen for scheduling and anti- or output dependences are violated, a temporary register is chosen, the instruction's destination is altered, flow dependent instructions sources are updated, and the dependences on the writer are corrected to reflect the new register lifetime relationships, as described above.

Selection of the lifetimes to rename can be tricky and a number of factors must be considered. First, the semantics of the move-commit instruction require that the full latency of the renamed instruction expire before the commit can take place. Unlike standard commits which essentially convert speculative writes into nonspeculative writes by indicating a guarantee of execution, move-commits actually transfer a value from one register to another. Therefore, renaming and speculating a long latency instruction up only a few cycles may actually lengthen the schedule by delaying the finish of the speculated instruction's home block until after its long latency has expired. Second, renaming should be applied wisely due to the limited number of registers available. Frequently, instructions deep in the trace that have no consumer in the trace are renamed and speculated to near the top. Generally, there is little benefit to this

type of speculation and so only instructions with consumers in the trace should be renamed and speculated.

6.4.5 Scheduler Table example

Figure 6.13(a) depicts the Scheduler Table utilizing the code example in Figure 6.4. Figure 6.13(b) depicts the state of the scheduler after the first three cycles have been scheduled. At this point instruction entries 0, 1, and 3 have already been scheduled and finished, and entries 2, 4, and 6 are ready. Suppose, due to its longer latency and higher priority, that the load in entry 6 is chosen to be issued next. Once issued, the command to free up antidependences on entry 6 is broadcast. However, while it appears that the move in entry 8 may now be free of antidependences, another read by the branch in entry 5 must still prevent it from being scheduled. By using the chain of reads constructed during the enqueuing process in the read link field, 8's new antidependence on 5 can be preserved. A free-or-update-antidependence(6 to 5 on `r2`) command must now be broadcast. Likewise, if 6 were in the middle of a chain of read dependences, then the read links that point to 6 must be updated to point to 5.

The Scheduler Table used in the experimental section of this work contains 256 entries. Each entry contains the 32-bit instruction, and 10 bits for home block identifier, 6 bits for finish time, and 8 bits for representing each dependence. The number of dependences in each entry depends on the number of explicit and implicit operands used by the instruction. Supposing a maximum of 8 operands (including memory), 2 fields of 8 bits each, requires 22 bytes per entry, for a total of about 7.5 KB storage. Unlike many software schedulers, linked lists are only used to link the previously mentioned read dependences. Instead, the mechanism relies upon

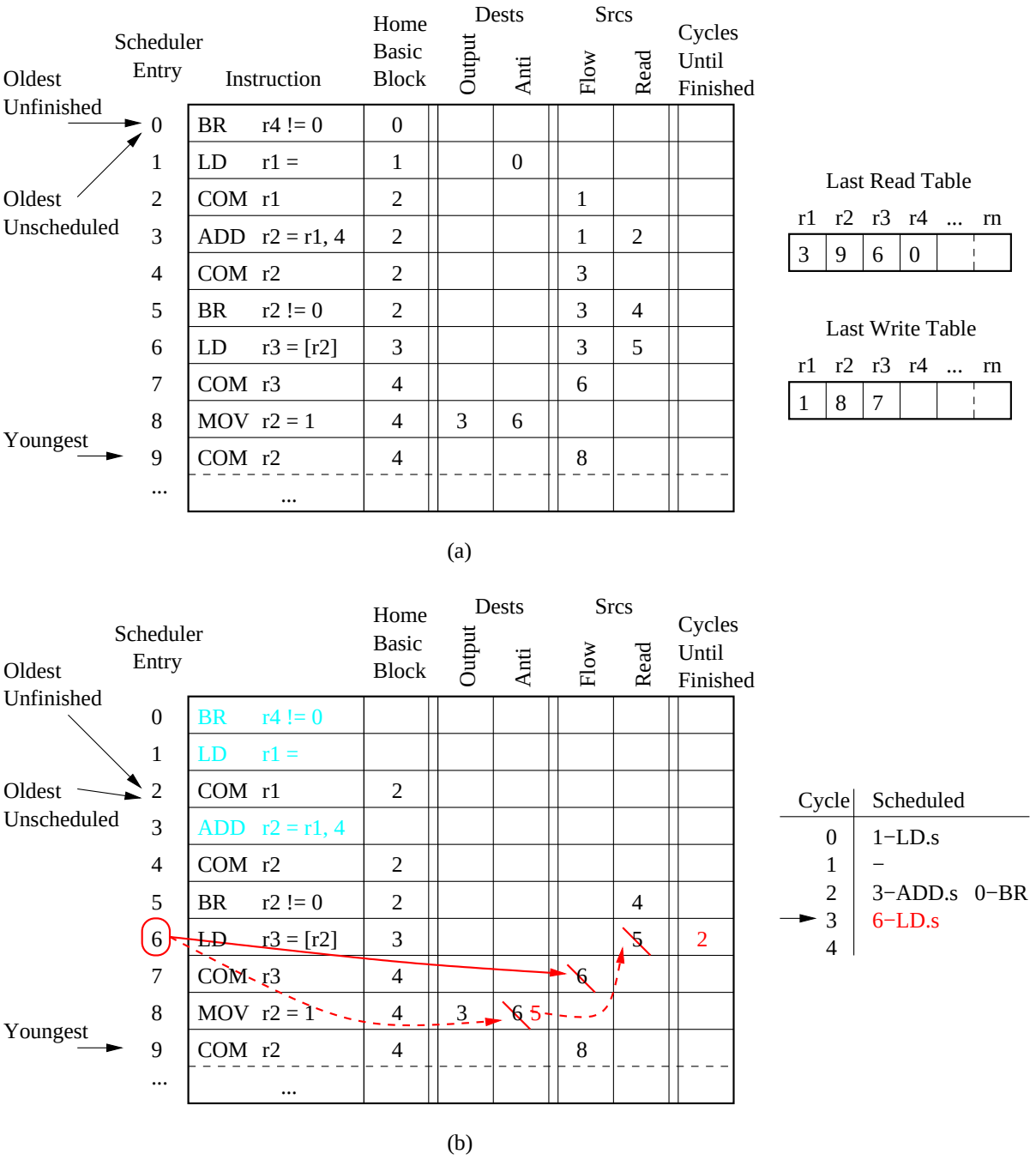


Figure 6.13 Scheduler Table hardware, assuming only one source and one destination operand. (a) After enqueuing instructions. (b) After scheduling first three cycles. Free flow dependence on entry 7. Update antidependence in entry 8 on entry 6 to entry 5.

the hardware's ability to perform many operations in parallel. Instead of freeing dependences one-by-one as stored in a list inside a software scheduler, this hardware mechanism relies on the broadcast messages to force a highly parallel table search. However, the ability to free dependences from anywhere in the table requires a large number of comparators. Unlike on-line schedulers, the off-line schedulers can afford several cycles to schedule an instruction. To process a command, each entry may need to search its operand fields for a matching entry. One implementation may consist of a comparator attached to each entry, requiring multiple cycles to search the operand fields of the entry. Furthermore, the scheduling selection logic could be implemented as a small microcoded engine, thus providing flexibility and upgradeability.

6.4.6 Instruction scheduling using store sets

During trace scheduling, it is critical to limit artificial dependences in order to provide maximal scheduling freedom. While store sets provide excellent memory resolution within a function, the identifiers have only intrafunction scope; they do not imply direct memory relationships between operations in different functions and are said to be incoherent. This phenomenon can be seen in Figure 6.1(c) where the store set on the store in `FN Y` is on a different store set than its consumer in the caller function `FN X`.

Figure 6.14 depicts the trace scheduling process for the example trace. During trace generation, function calls were inlined into the trace using the CALL_INLINE instruction, which was described in Section 5.5. As instructions are filled into the Scheduler Table, the last writer of each store set is maintained so that dependences can be drawn from subsequent loads to the latest store. The last writer array begins empty because early loads are not memory dependent

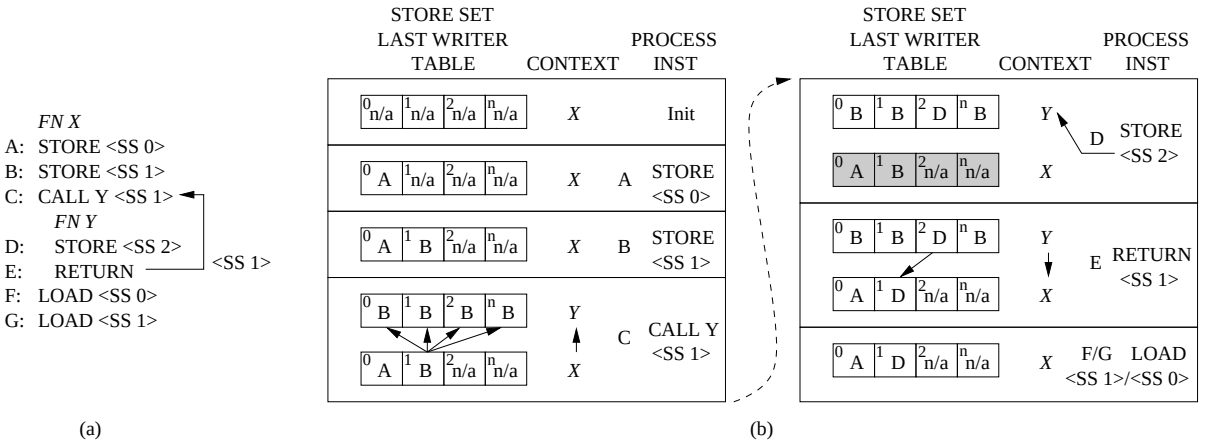


Figure 6.14 Maintenance of memory dependences during rescheduling through use of the Store Set Stack. (a) Example code sequence. (b) Operation of the stack.

on any store. As instructions A and B are processed, they are noted as the last writers of their respective store sets. When CALL_INLINE C is processed, a new last writer array is needed to track the store set accesses in the new function calling context. As previously mentioned, a callee may modify memory associated with the call's store set in the caller. From the callee's perspective, any of the loads in the callee may be dependent on the latest store in the caller that has the same store set as the call itself. Hence, when the new store set access context is created for the callee, all last writer entries are marked with the last writer of the call's set in the caller function. This process can be seen in Figure 6.14 when scheduling instruction C. Likewise, on a return, the last writer in the callee function may write the store set associated with the original call instruction. Hence the last writer in the callee will be the new last writer in the caller context, as can be seen in Figure 6.14 when scheduling instruction E. Note that load F on store set 0 will correctly find store A as its last writer (no writers due to the callee), while load G will be dependent on store D in the caller.

To maintain separate contexts for each inlined function, the ROAR architecture contains a stack of store set last writer arrays to manage the coherency problem. Because of implementation limits on the number of levels in the stack, it actually can be implemented as a circular queue of arrays, overwriting older contexts as necessary. However, when returning to any previously overwritten and hence invalid context, all caller store sets must be conservatively written with the last writer in the callee instead of just the call's store set. In this work, memory operations are assumed to contain their store set identifiers within the instructions themselves, although a convenient table-based approach might also be possible.

6.5 Instruction Scheduler Limitations

The current design of the scheduling mechanism does not take full advantage of predication. First, the scheduler cannot speculate predicate definitions across function boundaries. To use a register speculatively in a particular cycle, the speculative version of the register must be free, and there must be no reads or writes of the nonspeculative version between the target cycle and the original location of the speculative instruction. Predicate definitions near the beginning of functions are bounded by the predicate register file spill (read of all predicates) at the top of the function, and predicate definitions immediately after an inlined return are bounded by the predicate register file fill (write of all predicates) at the end of the callee. One solution would be to make the spill and fill instructions only read and write the nonspeculative versions of the predicates, thereby allowing the speculative sides to contain speculative values.

Through predication, two register writes may occur to the same register in the same cycle if they are on different predicates. The current framework implementation, however, analyzes these two writes as output dependent and places them in different cycles. In this situation, it is possible for the ROAR scheduler to lengthen the schedule created by the compiler. By analyzing the incoming bundling, this false output dependence could be avoided and the independence of the two predicates could be determined.

6.6 Instruction Scheduling Experimental Setup

To test the effectiveness of the ROAR system, a number of experiments were conducted. An EPIC platform was chosen for these studies to better examine the amount of instruction-level parallelism that could be exploited in the applications under test.

6.6.1 Benchmarks

Listed in Table 6.1 are the applications used to test the performance ROAR. They represent a wide variety of application types selected from SPEC CPU95 [12], SPEC CPU2000 [12], and MediaBench [39]. Each application was compiled through the IMPACT ILP Compiler (Internal Version 03-18-2002) [40], [41], [42], [43] using its distributed training inputs to generate basic block profile information. The applications were then evaluated using selected inputs that simulate in a reasonable amount of time and which represent various levels of similarity to the training inputs. The inputs include the distributed test inputs (which were designed to be a subset of the training inputs), training inputs themselves, distributed reference inputs (which were

Table 6.1 Benchmarks and inputs used in SHOP experiments.

Benchmark	Inputs	Baseline CLS Dyn. Inst (millions)	Baseline CLS Cycles (millions)	Baseline ILP Dyn. Inst (millions)	Baseline ILP Cycles (millions)
124.m88ksim	1 - train	67.1	58.9	69.8	38.4
130.li	1 - train	112.3	115.9	131.7	73.6
	2 - reduced test (6 queens)	28.3	29.0	35.8	17.3
	3 - reduced ref (au browse xit)	335.7	352.4	445.7	245.8
132.ijpeg	1 - train	1105.5	568.6	1106.1	336.4
	2 - new (dither faces)	55.8	34.2	58.2	27.2
	3 - new (lower quality scenery)	321.5	257.3	330.0	223.6
134.perl(123) (compiler train on 1, 2 and 3)	1 - train jumble	1363.8	1302.0	1433.7	947.5
	2 - train scrabbl	26.2	23.6	30.3	17.0
	3 - train primes	7.8	6.1	8.8	4.4
134.perl(23) (compiler train on 2 and 3)	1 - train jumble	1939.0	1551.8	2301.1	1456.6
	2 - train scrabbl	25.8	22.6	30.5	16.4
	3 - train primes	6.6	5.5	7.7	3.8
164.gzip	1 - reduced train	1756.4	1688.8	2040.2	1027.5
175.vpr	1 - test	991.3	1117.1	1099.5	671.8
	2 - test	398.9	441.6	434.5	335.0
181.mcf	1 - test	105.0	228.7	120.5	135.9
197.parser	1 - UMN small reduced ref	178.6	204.4	186.3	150.1
255.vortex	1 - UMN medium reduced ref	182.0	174.9	162.5	74.6
	2 - UMN large reduced ref	520.2	492.9	466.4	216.3
300.twolf	1 - UMN small reduced ref	49.8	66.0	62.2	39.4
	2 - UMN large reduced ref	519.9	677.0	625.0	399.2
mpeg2dec	1 - train	97.4	71.9	81.1	49.3
wc	1 - 16 K-line C source file	1.1	0.8	1.1	0.6

designed to be similar to but a superset of the training inputs), reduced reference inputs (which are either denoted as such or were designed by the University of Minnesota (UMN) [44], [45] to be shorter in length than the distributed reference inputs but maintain similar or subset coverage), or new inputs altogether.

6.6.2 Compiler

As previously mentioned, the IMPACT research ILP compiler built each of the applications from native C code in order to provide a solid, aggressive code base for which to examine ROAR system performance. The compiler consists of a number of modules summarized in Figure 6.15. From C code, the preprocessed applications proceed through IMPACT's highest level intermediate representation called *PCODE*. While in this abstract syntax tree representation, basic block control-flow profiling is performed that guides cross-file function inlining. After inlining, aggressive interprocedural pointer aliasing analysis [31] generates memory dependence information used throughout the compilation process. Once analyzed, the application is converted into an instruction-level intermediate representation called *LCODE* that targets a generic EPIC processor. On this representation, a variety of architecture-independent optimizations can be applied including classical optimizations, hyperblock predicated region formation utilizing advanced predicate analysis [46], superblock formation and optimization [47], and other ILP-enhancing optimizations. Then the application is transformed into an architecture-dependent variant of LCODE called *MCODE*. On this representation, peephole optimizations are performed followed by two rounds of instruction scheduling utilizing sentinel control speculation [33], and register allocation. Furthermore, memory dependences in each function are analyzed and collected into the maximum number of independently accessed memory regions which are then round-robin assigned to the available store set identifiers. Memory operations and call instructions are subsequently annotated with their appropriate store set.

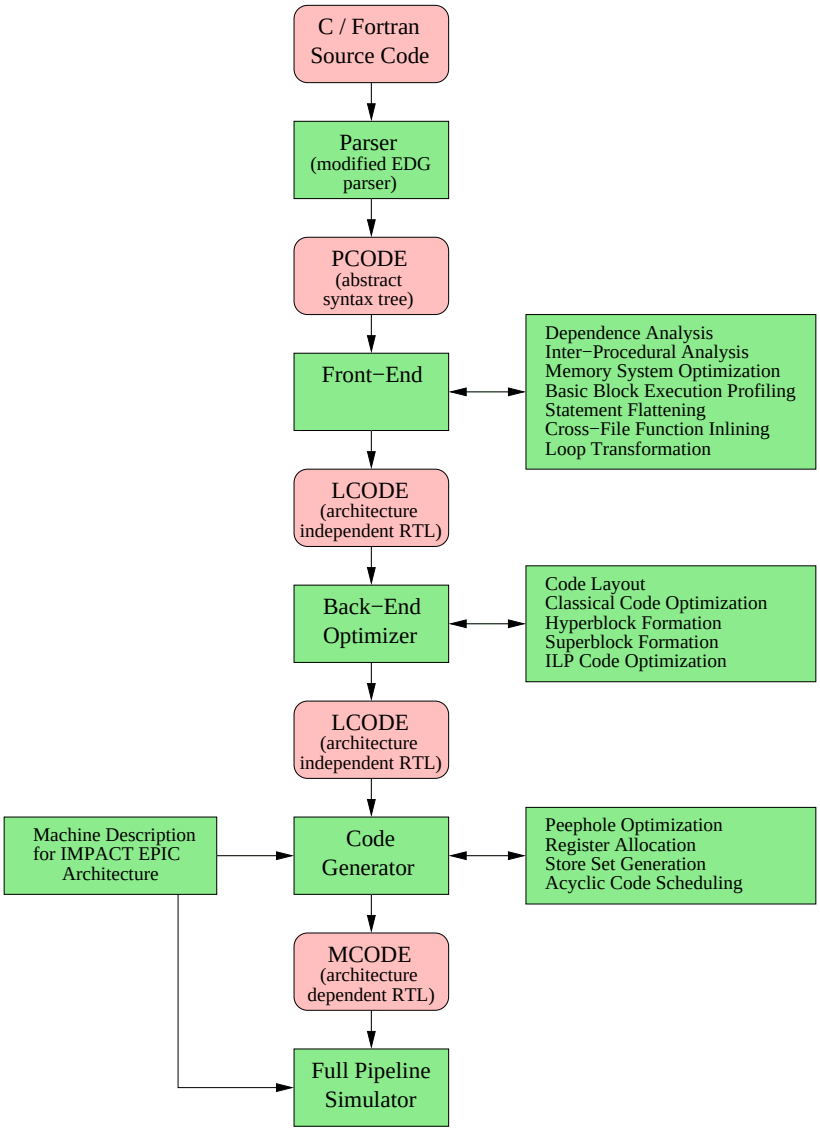


Figure 6.15 The IMPACT compiler.

The experiments in this chapter utilize two levels of compile-time optimization aggressiveness to demonstrate ROAR's benefits. At the first level, the compiler applies only classical optimizations to the applications (*CLS* codes). At the second level, the compiler applies hyperblock and superblock formation along with other ILP optimizations (*ILP* codes). Select portions of the C library, mainly portions of the string and memory manipulation and sorting

functions, were also compiled through IMPACT with both levels of aggressiveness but were not inlined into the applications in order to simulate dynamic library linking.

6.6.3 Simulator

The performance measurements reported in this work are generated by a custom software simulator, described in Figure 6.16, that performs cycle-by-cycle full-pipeline simulation of each instruction. The simulator fully accounts for the effects of branch prediction, wrong path execution, cache utilization and pollution, varying memory latency, interlocking, and by-passing. Since the applications are tested within the simulation environment, the TGU and scheduler generate real traces on-the-fly that are actually used during execution, thus verifying the trace generation and optimization techniques.

The simulator is essentially a virtual machine that executes instructions on behalf of the simulated application and performs services, such as I/O, for the application. As shown in Figure 6.16, the MCODE intermediate representation of the application and subset of the C library are loaded in the simulator, which proceeds to map the instructions and data to actual memory locations in reserved spaces that correspond to loaded sections of a binary. The data space serves as the actual data locations for the application, but the instruction memory only serves to provide instruction address for cache simulation purposes; instruction are fed to the interpreter in their MCODE representations. When the simulated application requires input or output, the requests are simply passed to the input and output routines of the simulator, essentially making the simulator a transparent virtual machine layer. When a function call is made or a data element is accessed, the MCODE representation only contains a name for the

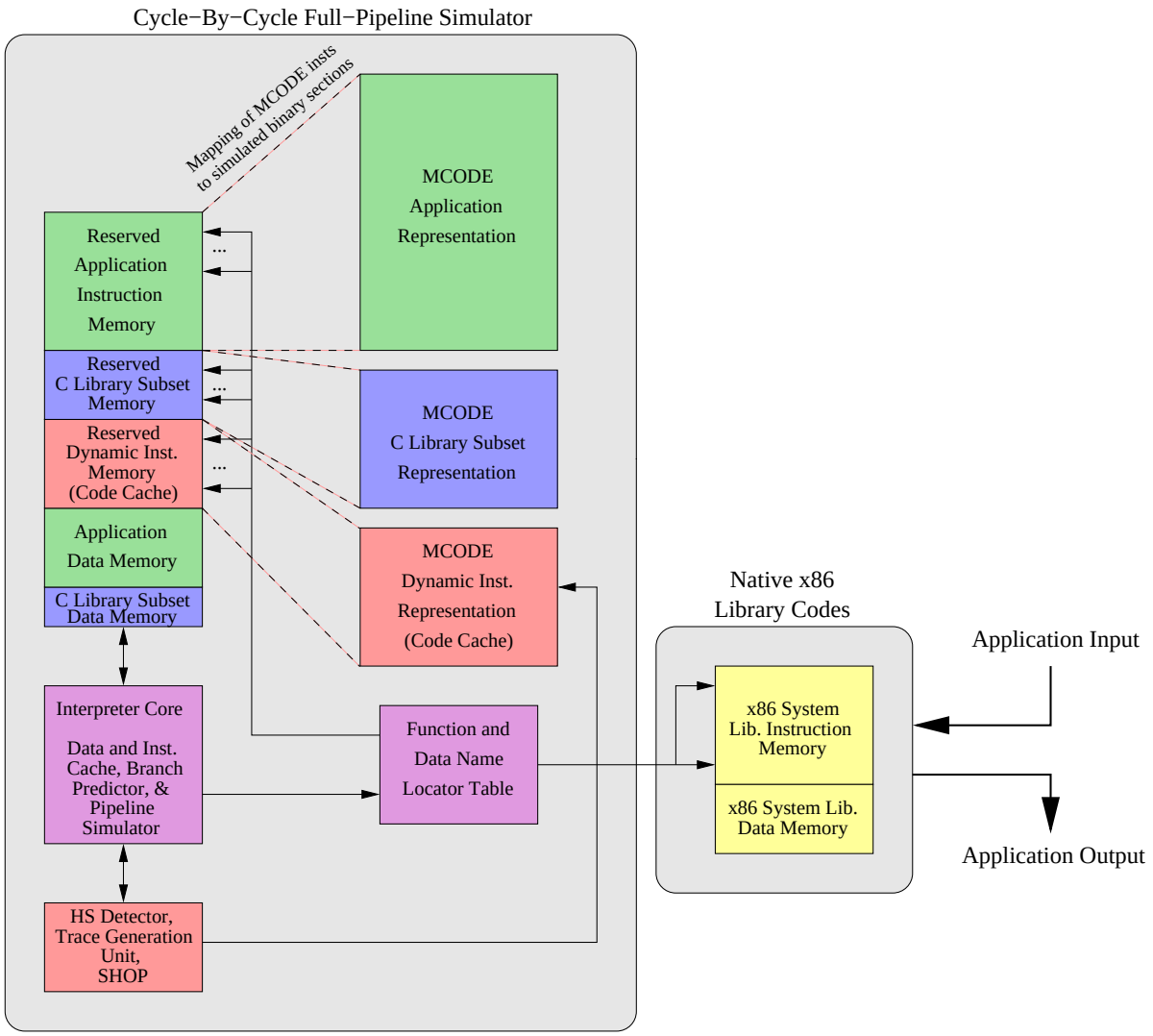


Figure 6.16 Simulator framework.

function or element. Hence, it is the responsibility of the name locator table to translate each name into its mapped address, or to identify it as a system call located outside of the simulator. As previously mentioned, a subset of the C library calls have been compiled and included with the application MCODE, thus enabling cycle-by-cycle simulation of these codes as well. A number of library calls and system services cannot be compiled through IMPACT with current

compiler constraints, and thus the native library codes are called on behalf of the application. Cycles spent in the native calls, therefore, are not accounted for in the simulation results, but generally represent an insignificant portion of overall application execution.

6.6.4 ROAR architecture

The architecture modeled consists of a 10-stage pipeline containing five functional unit types (Integer ALU, FP, Long Latency FP, Memory, and Control). The simulations include a multilevel memory hierarchy, branch and return address predictor, Branch Behavior Buffer, Trace Generation Unit, and SHOP trace scheduler. Table 6.2 reflects the architectural parameters chosen for the evaluation system. Table 6.3 enumerates the parameters of the Hot Spot Detection hardware. These parameters are derived from previous experiments [2] and are tuned slightly for EPIC codes. Based on observations described in Subsection 5.6.2, the BBB was enlarged to account for more branches while the candidate ratio (candidate threshold divided by refresh timer interval) was lowered to account for the branch replication effects of loop unrolling, tail duplication, etc. It is important to again note that the results account for a detection to rescheduled code delay, which can surpass 100 000 cycles. Additionally, the first fetch of any new optimized hot spot code takes the full L2 miss penalty of 50 cycles.

6.7 Instruction Scheduling Evaluation

To evaluate the effectiveness of the rescheduling system, a number of applications were evaluated. These experiments were designed to show the benefits of the ROAR scheduling

Table 6.2 Simulated EPIC machine model.

Instruction issue	8 units
Integer arithmetic and logic unit	5 units
Floating-point arithmetic unit	3 units
Memory unit	3 units
Branch unit	3 units
Branch predictor	10-bit history gshare, 3 predictions per cycle
Branch Target Buffer size	1024 entry
Return Address Stack size	32 entry
Branch resolution	7 cycles
Load/store buffer size	8 entry each
L1 data cache	64KB
L1 instruction cache	64KB
Unified L2 cache	512KB
Store sets	16 (4-bit encoding)
Integer registers	64
Floating-point registers	32
Predicate registers	64
Operands per commit	4
Operands per move-commit	1

Table 6.3 Hot Spot Detector, Trace Generation Unit, and Scheduler and Optimizer hardware parameter settings.

Parameter	Setting
Number of Branch Behavior Buffer sets	512
Branch Behavior Buffer associativity	4-way
Executed and taken counter size	9 bits
Candidate branch execution threshold	16
Refresh timer interval	8192 branches
Clear timer interval	65 536 branches
Hot Spot Detector counter size	13 bits
Hot Spot Detector counter increment	2
Hot Spot Detector counter decrement	1
Scheduler Table entries	256
Store Set Last Write Table contexts	40

Table 6.4 ROAR hot spot detection results for CLS and ILP codes with scheduling and renaming (S+R).

Benchmark	Input	CLS S+R: Num. of Hot Spots	CLS S+R: Code Cache %	CLS S+R: TGU Active %	ILP S+R: Num. of Hot Spots	ILP S+R: Code Cache %	ILP S+R: TGU Active %
124.m88ksim	1	5	52.8	1.8	7	76.9	3.2
130.li	1	11	66.4	1.4	11	61.1	2.3
	2	6	63.8	4.1	8	68.0	4.9
	3	13	75.4	0.7	8	67.4	1.8
132.ijpeg	1	10	82.1	0.3	8	64.9	0.3
	2	16	72.9	9.7	14	73.1	10.3
	3	13	83.8	0.7	13	73.8	1.0
134.perl(123)	1	9	77.4	0.1	11	71.8	0.2
	2	3	55.1	2.2	3	50.6	2.3
	3	5	47.2	11.7	5	50.2	13.2
134.perl(23)	1	9	68.6	0.1	12	86.6	0.1
	2	3	54.7	2.4	4	46.5	2.8
	3	3	60.4	8.7	4	57.6	13.4
164.gzip	1	23	84.5	6.5	19	81.5	1.6
175.vpr	1	5	49.3	0.1	6	65.4	0.2
	2	19	88.7	1.1	23	72.4	2.7
181.mcf	1	20	72.4	2.5	23	75.0	4.4
197.parser	1	26	53.3	2.7	19	48.5	2.4
255.vortex	1	12	58.0	1.5	7	51.7	2.4
	2	34	39.5	1.4	9	43.2	1.0
300.twolf	1	3	47.1	1.7	3	56.1	3.2
	2	12	63.0	0.6	11	52.3	0.9
mpeg2dec	1	4	84.1	0.7	2	79.5	1.0
wc	1	1	84.1	9.8	1	54.0	51.1

framework. Clearly, further optimization possibilities exist that may provide further performance enhancements.

6.7.1 Performance of optimized traces

Table 6.4 details the total number of detected hot spots, the percentage of instructions executed from the code cache, and the percentage of instructions in which the hot spot detection

and formation hardware were active for both CLS and ILP codes. The results show general similarity between the results of CLS and ILP codes (*130.li*, for example). Slight variations in the number of hot spots detected are expected and observed due to variations in detection timing with regard to the beginning of the program phases. The TGU is typically active for less than 5% of the dynamic instructions for longer running applications; it, however, can sometimes be active for a much larger percentage when the number of dynamic instructions is within an order-of-magnitude of the reset interval (*134.perl* input 3, for example).

Figure 6.17 shows the cycle accounting measurements taken for each benchmark and input. These results were generated by older IMPACT ILP Compiler and ROAR versions (Internal Version 11-18-2001). While the relative performance of each configuration varies somewhat from the overall results presented next, the trends remain the same. For each processor cycle, a determination is made as to why further instructions could not be dispatched from the front-end to the back-end of the pipeline in that particular cycle. The cycles are distributed into several different categories: *Fetch* (front-end stalls due instruction cache misses and taken branch penalties), *Stop Bit* (explicit dispatch breaks), *Oversub FU* (pipeline backup due to oversubscribed functional units), *Wrong Path* (execution cycles entirely wasted due to mispredicted branches), *LD Dep* (stalls due to unsatisfied flow dependences from load instructions), *Other Dep* (stalls due to flow dependences from nonload instructions), and *Unstalled* (unrestrained dispatch of instructions). The benchmark input number is listed below each run where the four bars depict the cycle accounting normalized to baseline CLS code for baseline CLS, CLS with SHOP scheduling, baseline ILP, and ILP with SHOP scheduling.

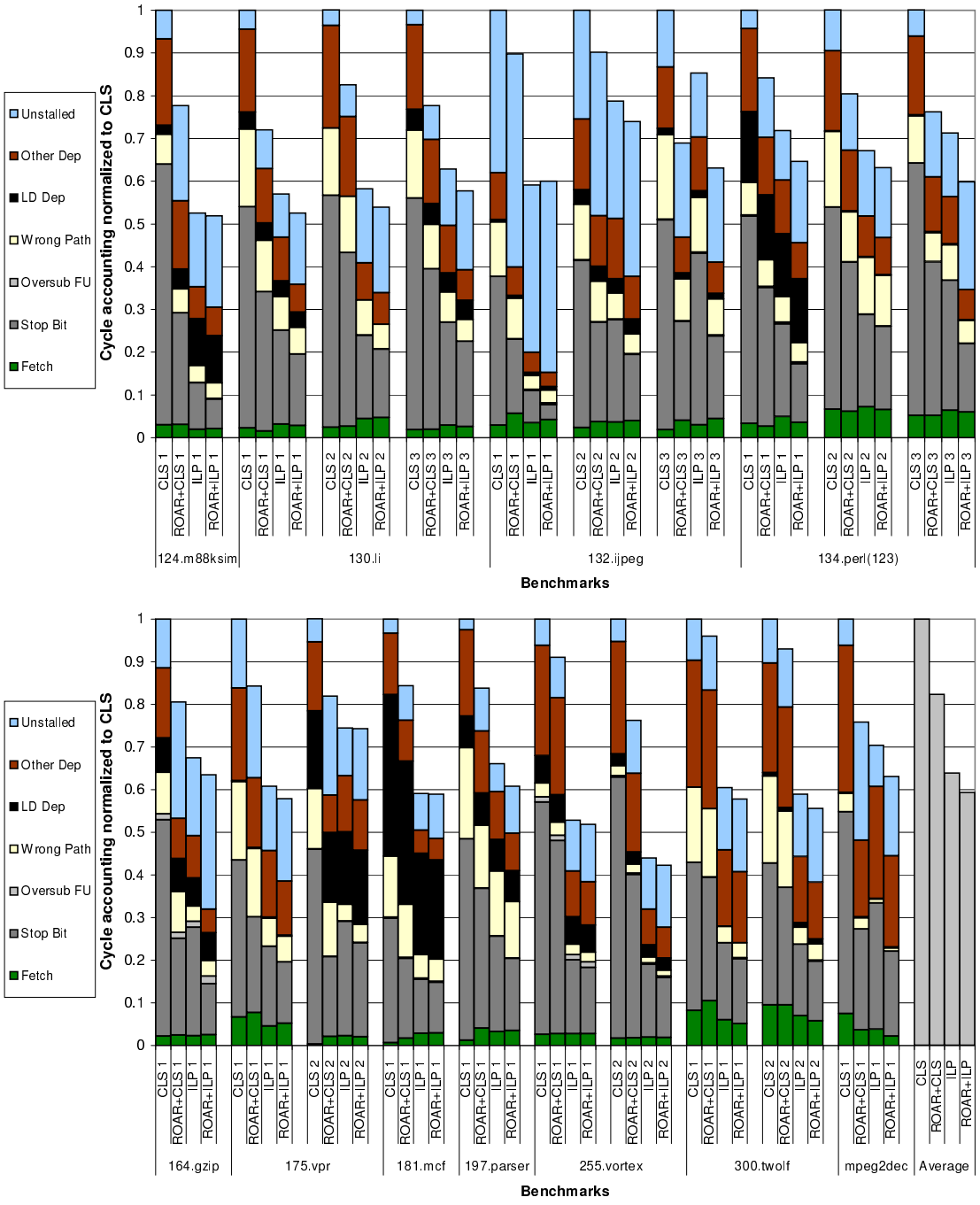


Figure 6.17 Cycle accounting normalized to baseline CLS code for baseline CLS, CLS with SHOP scheduling, baseline ILP, and ILP with SHOP scheduling.

As shown in figure 6.17, much of the reduction in cycles comes from a reduction in the number stop bits reached. This implies that ROAR has a more effective use of the machine width because fewer cycles are ended because of a stop bit even though more instructions await execution. This is particularly evident in *134.perl* input 3. Also notice the typical increase in unstalled cycles, which corresponds to the above observation.

Six experiments were conducted for each input of the benchmarks to gauge overall performance due to the TGU and SHOP, the results of which are shown in Figure 6.18. The five data bars represent speedups over CLS baseline codes for TGU traces for CLS codes, TGU traces with scheduling and renaming for CLS codes, baseline ILP codes, TGU traces for ILP codes, and TGU traces with scheduling and renaming for ILP codes. In general, TGU trace generation alone provides little benefit since it only improves front-end fetch performance but does not affect the minimum number of cycles required for the application to proceed through the back-end of the pipeline. Out-of-order execution cores can benefit, however, from increased fetch performance through speculative out-of-order execution. Slight degradations due to TGU traces alone can be attributed to the instruction cache; it often experiences more misses because the total working set of instructions (side exit and non-phase portions of the original code plus the trace sets) is typically somewhat larger than the working set of just the original code. For example, the TGU provides a speedup of 9% for the CLS code for *mpeg2dec*, while proving a slowdown of 2% for the ILP code compared to ILP baseline. However, the SHOP provides an additional 15% speedup from the TGU traces (totaling 24%) for CLS code and an additional 16% speedup from the TGU traces (totaling 14%) for the ILP code all over CLS baseline. This

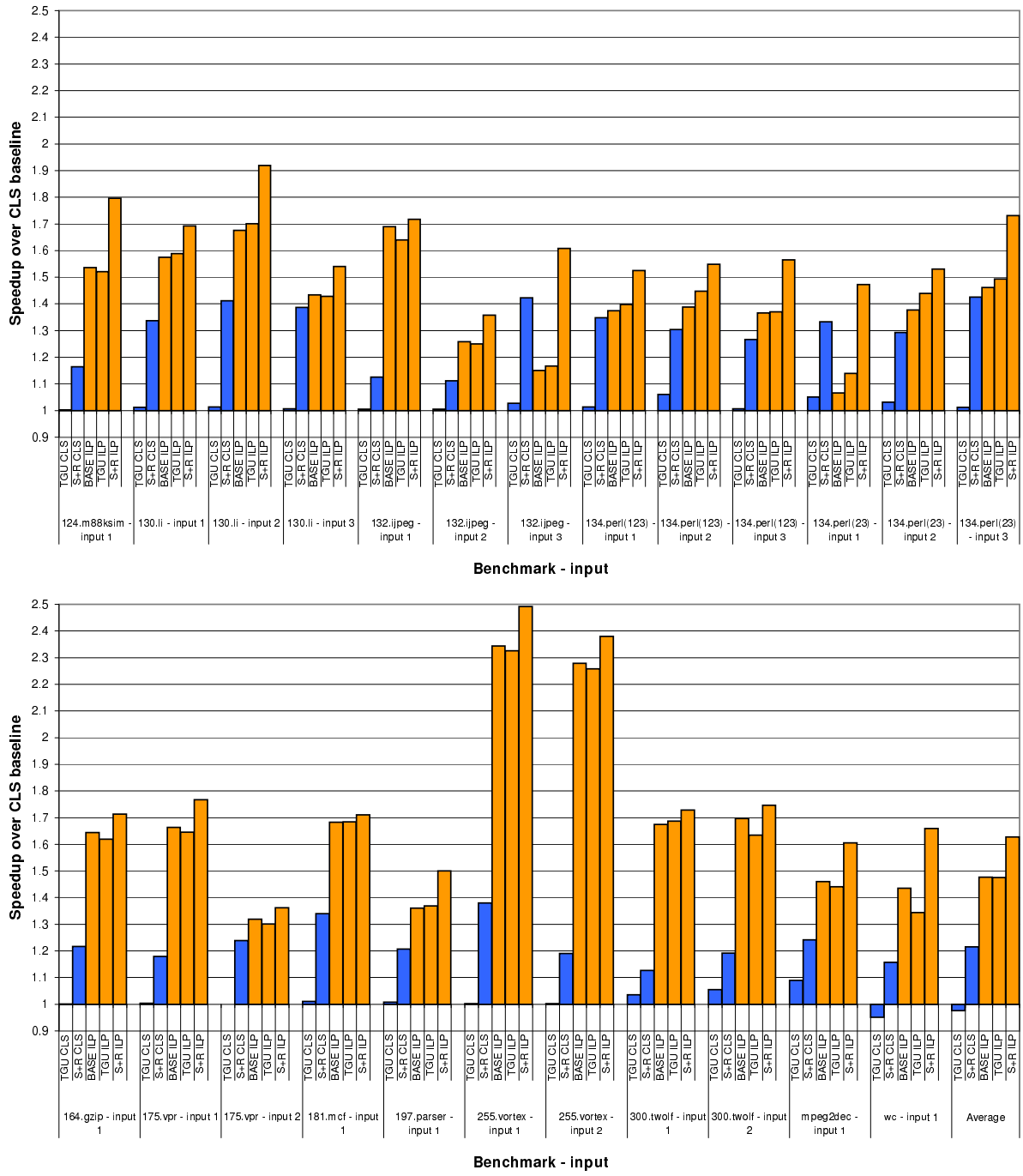


Figure 6.18 Speedups for various hardware configurations over CLS baseline.

translates to a speedup of about 10% for SHOP on ILP code over ILP baseline. The results for *mpeg2dec* closely resemble the overall average.

6.7.2 Benefits of store sets

To evaluate the benefits achieved through utilization of compiler-annotated store set information, CLS and ILP code was simulated with ROAR optimization for four benchmarks using 1 (no set differentiation), 2, 4, 16, and 1024 store sets. The overall speedups for both CLS and ILP codes due to instruction scheduling with renaming and copy propagation are shown in Figure 6.19. The performance is alternatively shown in Figure 6.20 for CLS and ILP codes as a fraction of the speedups of CLS-1 and ILP-1 (no store set differentiation) over baseline, respectively. For most benchmarks, 16 store sets are sufficient to achieve adequate disambiguation.

In *255.vortex*, function calls on a particular store set were often preceded by a write to that store set and followed by a read from it. This scenario is commonly found when global or heap memory is updated prior to the function call, modified within the function call, and read after the call returns. Because the function may modify the same location as the prior write, both the call and the prior write will be assigned the same store set. As previously described, this situation sets up conservative aliases due to store set coherence between the prior write and all accesses inside the function call. Thus, all loads within the function are pinned below the prior write. Similarly, any write within the function call will appear as a modification to the same set as the subsequent read in this scenario. Thus, reads following the return are pinned below the last write inside the function call.

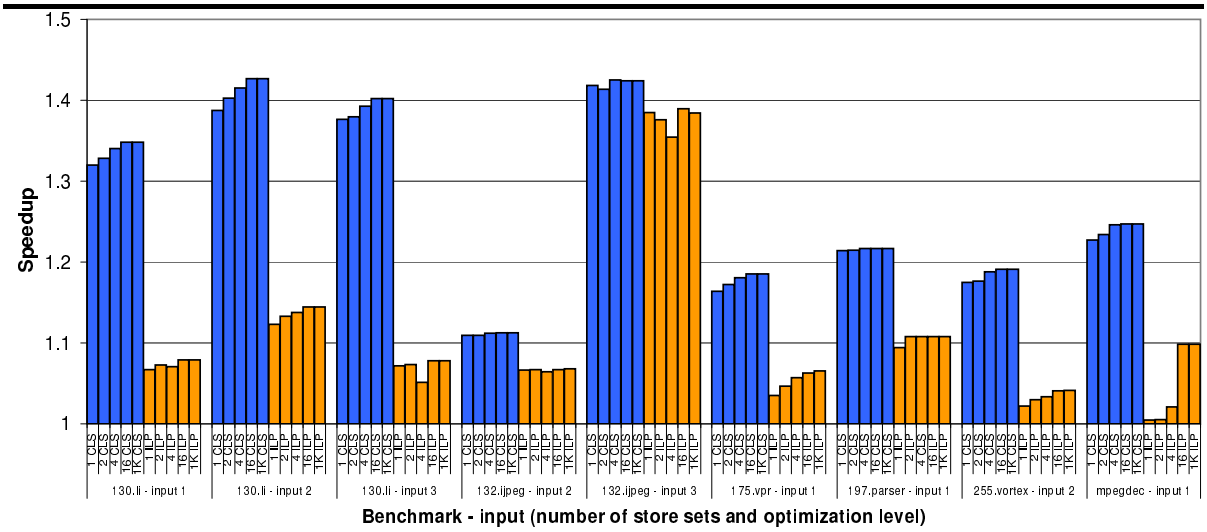


Figure 6.19 Speedup for 1, 2, 4, 16, and 1024 store sets for CLS and ILP codes.

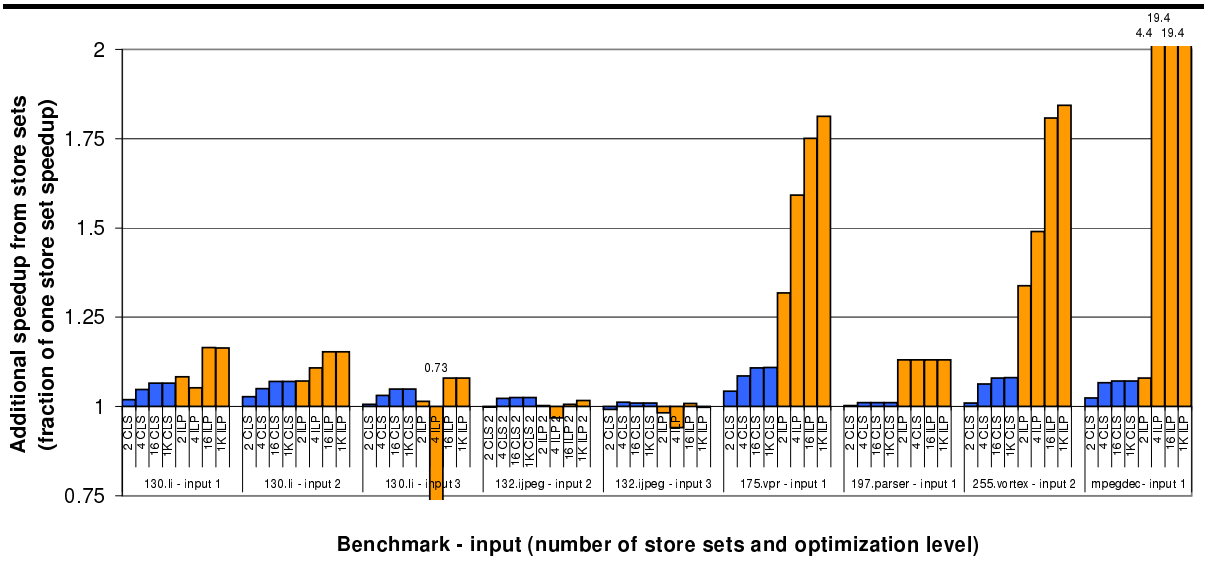


Figure 6.20 Additional speedup for 2, 4, 16, and 1024 store sets as a fraction of 1 store set speedup for CLS and ILP codes.

A number of other factors have been linked to the modest performance of memory disambiguation through store sets. First, many register spill instructions surround function calls, both for caller-saved registers just prior to the call instruction and callee-saved registers at the beginning of the called function. Hence, several cycles are often consumed with fully utilized memory functional units. Subsequent loads must be moved up above the block of stores, or mixed in with the stores. However, the scheduling of speculative loads into this block may lead to delayed scheduling of stores and branches, thus unduly penalizing later side exits, similar to what was described in the instruction priority discussion in Subsection 6.4.1. These groups of stores also often save registers that will be overwritten later in the function call. Therefore, it is likely that loads and other instructions in the called function will use these saved registers. They will be prevented from moving above the stores that save their original values. It is, however, possible to rename such instructions.

Stack loads also suffer from an additional restriction because of their stack pointer operand. Since the stack pointer register is updated on call and return instructions, caller-save register fills following the return of a function call cannot be moved up into the called function. In general, stack operations are pinned inside of their own function due to their dependence on the stack pointer register. This situation motivates an optimizer and scheduler extension that would allow for pre-schedule-time register renaming. For example, the new stack pointer value for a called function could be computed and placed early in the trace. Then, stack pointer register operands inside the function call could be reassociated to the new temporary register. This would allow for stack load speculation above the call instructions.

Out-of-order microarchitectures also have an advantage over in-order microarchitectures because they can dynamically determine or predict a memory dependence, whereas compilers have to be conservative and mark all possible dependences. Data speculation [48], [49] is useful with in-order machines like EPIC for memory operations that could alias but rarely do. Essentially, loads are allowed to move above stores that they may alias with, but a check must be inserted at the original site of the load. This check verifies that none of the stores actually wrote to the loaded memory location. A failed check triggers re-execution of the load and its dependent instructions. Data speculation could also be adapted to operate under the Precise Speculation model to allow ROAR to safely reorder loads and stores that have the same store set when they alias infrequently.

Lastly, much of the store set support in ROAR is a result of utilizing intraprocedural store sets. Again, the motivation was to provide more detailed coverage within a function by allocating all of the store sets to each function body and then providing a coherence mechanism. Future work should examine carefully allocated global store sets which will not have the coherence problem.

6.7.3 Benefits of scheduling and optimization

Figure 6.21 depicts the speedups of CLS and ILP codes over their respective baselines due to various levels of optimization for a representative subset of the benchmarks. The left bar in each set depicts the performance of just the remapped traces. On average, the performance is nominally the same as the baseline. The code straightening optimization performed by the TGU has less of an impact on in-order cores since they do not speculatively execute

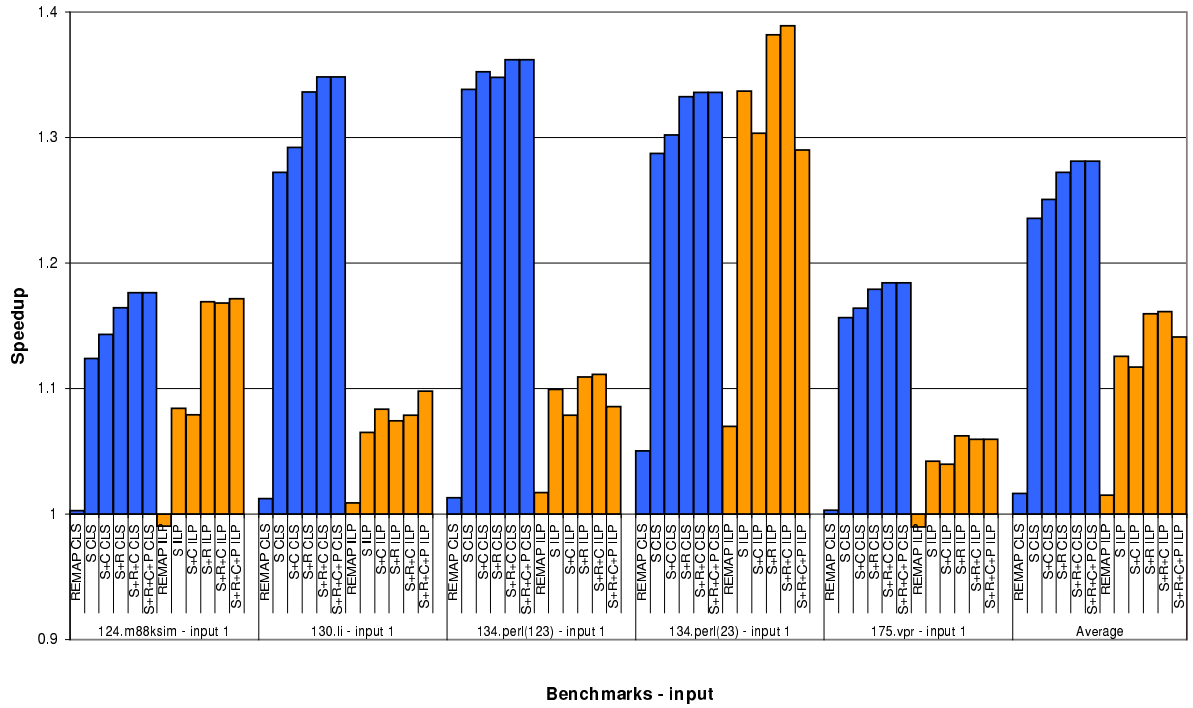


Figure 6.21 Speedups due to various levels of optimization. REMAP: TGU generated traces only. S: Scheduling. C: Copy propagation. R: Register renaming. P: Early path splitting optimization.

instructions out-of-order from the predicted path. The next bar represents the addition of instruction scheduling which is the bulk of performance improvement (23.5% for CLS codes and 12.5% for ILP codes). Instruction scheduling exploits the new block layout by speculating instructions from successive blocks to earlier positions in the trace. The impact of scheduling is more significant in CLS codes than ILP codes since aggressive code layout optimization in the form of superblock formation [47] has already been applied to ILP codes by the compiler.

The next bars represent scheduling with forward copy propagation, scheduling with renaming, and scheduling with forward copy propagation and renaming. Forward copy propagation sometimes hinders performance by extending a particular register's live range, thus preventing

a successive live range in the same register from being speculated. The last bar represents the addition of an experimental predicate optimization, called *early path splitting*. When callee instructions are inlined following a predicated call instruction, execution of the call with a false predicate (fall through the call) will force flow back to original code following the call. In between the call and the defining predicate are a number of parameter moves that are predicated on the same predicate as the call. The early path splitting optimization inserts a jump immediately after the predicate definition to original code for the false predicate. Once in place, the predicates for the call and parameter moves can be converted to true, thus enabling copy propagation. This optimization has no effect on CLS codes since then contain no predication. The resulting performance due to this optimization on ILP codes is mixed because execution flow may exit from optimized code much earlier when the predicate is false. Overall, an average of 28% speedup is achieved for CLS codes with all optimizations, while an average of 16% is achieved for ILP codes with all but the path splitting optimization.

6.7.4 Trace optimization example: *134.perl*

Figure 6.22 depicts an example of ROAR's inherent ability to perform multilevel partial inlining. An original trace of execution, taken from the sixth hot spot of *134.perl* input1, is presented in Figure 6.22(a), assuming perfect branch prediction and caching. For each cycle (row), the instructions executed in that cycle are shown as blocks, where each color represents a different class of instruction. The first cycle, for example, contains memory, ALU, and fall-through branch instructions. The second set of columns depicts the same instructions correlated to their original three functions (note that the compiler inlined `qsort()` into `do_sort()`).

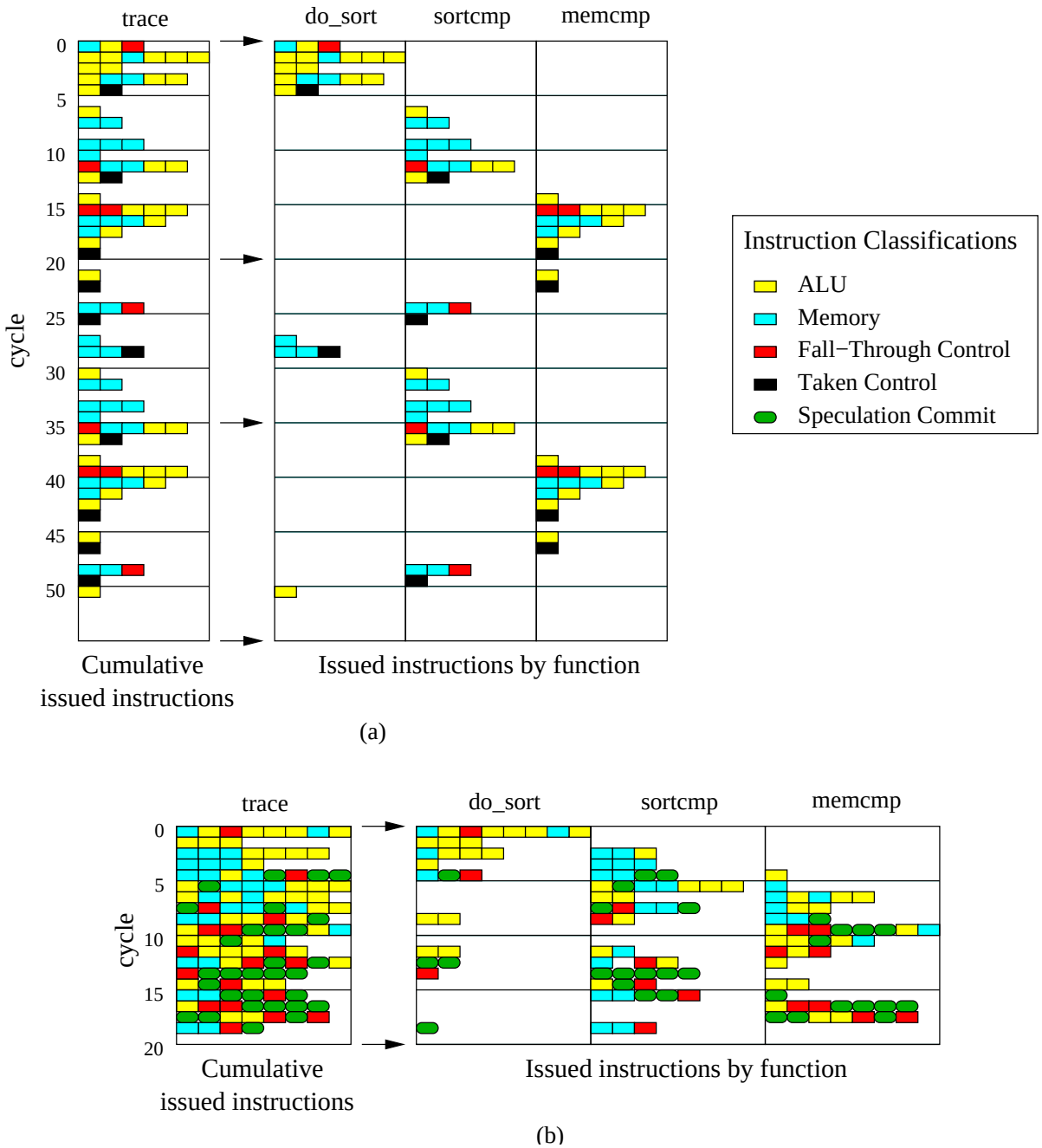


Figure 6.22 Optimization of a trace from *134.perl*. (a) Original execution flow. (b) Resulting execution after inlining, straightening, and rescheduling.

Like many architectures, the EPIC pipeline configuration causes taken branches to exhibit a one cycle taken-branch bubble, as shown by an empty cycle following a black, taken control-flow instruction. Figure 6.22(b) shows the same instructions within the detected hot spot trace after rescheduling. The ovals (green) represent the instructions that commit speculated values (renamed or not) to their nonspeculative registers. The resulting execution is approximately 60% faster. This trace benefits heavily from register renaming to eliminate the reliance on the parameter passing registers, and from store-set-based memory disambiguation, speculation, and renaming to boost load-dependent chains higher in the schedule.

6.7.5 Trace optimization example: *wc*

Due to the nature of the trace generation process, the TGU inherently performs a partial iteration loop peeling optimization. Loop peeling is the process of moving the first iterations of a loop out of the body and explicitly placing them in the loop prologue [50]. Moving a portion of the first iteration to the prologue and then reforming the loop essentially rotates the body so that the former bottom portion of the body is now at the top. Consider Figure 6.23, which depicts an example hot region from the *wc* microbenchmark. The original code sequence as generated by the compiler, presented in Figure 6.23(a), consists of three traditional basic blocks that each terminate in a branch. The top portion of the column represents the code in a basic-block diagram form while the bottom portion depicts the instruction schedule form. During the trace formation process, the various branches in the code and their targets are tracked. As the code is copied and straightened, branches `A`, `B`, and `C` are seen for the first time and placed into the trace, as shown in Figure 6.23(b). Note that while branch `C` actually branches to the

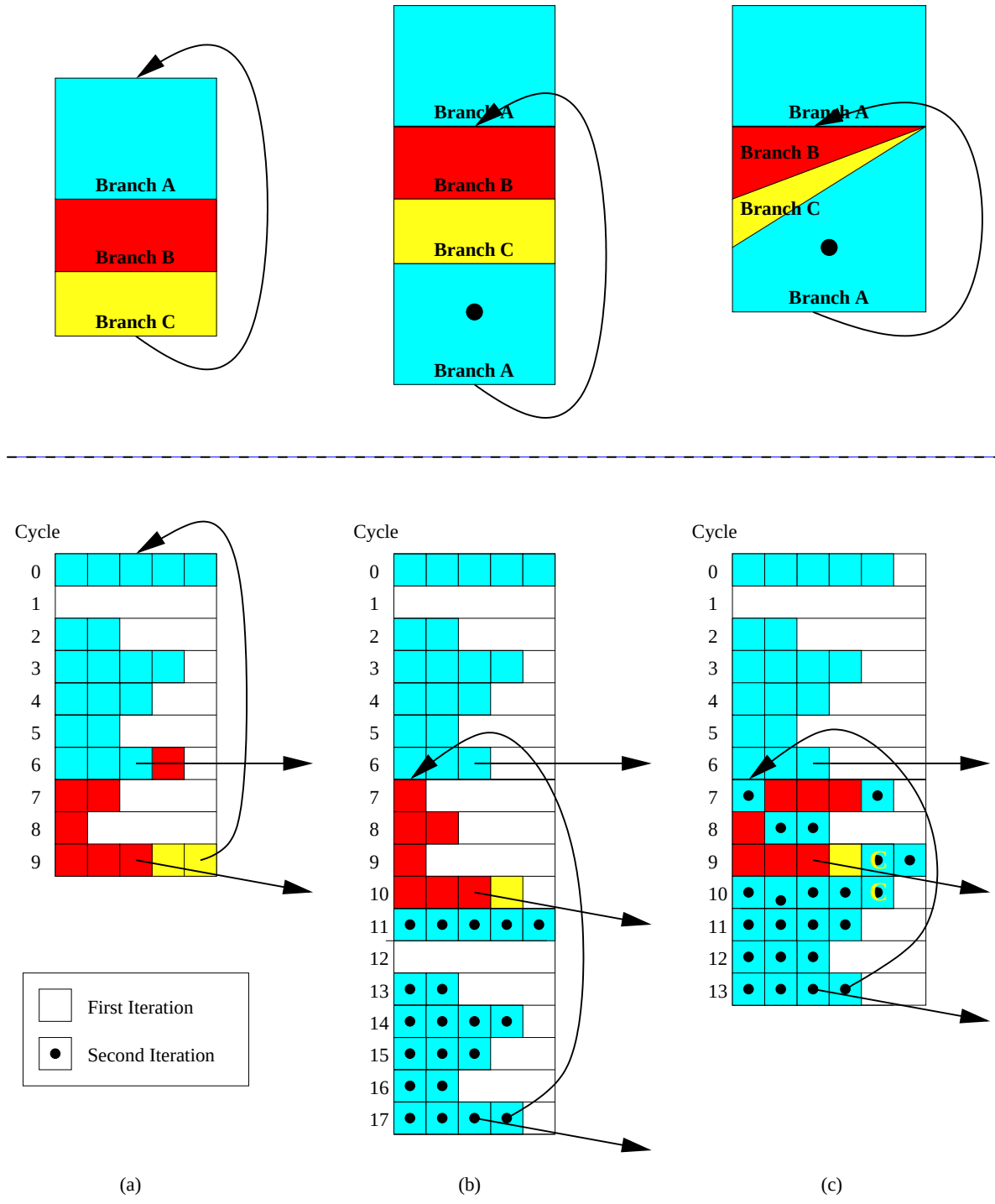


Figure 6.23 Optimization of a region from the microbenchmark *wc*. (a) Original loop with the three traditional basic blocks highlighted. Block diagram on top, instruction schedule on bottom. (b) Region after relayout. (c) Region after scheduling.

block terminated by branch A, branch C has not previously been seen and therefore its target is unknown. Therefore, relayout continues through a second copy of the block terminated by A. Once reaching branch A for the second time, its fall-through path has already been remapped, and so the TGU inverts the branch and links it to the fall-through target. Note that the loop body now consists of the blocks associated with B, C, and A, in that order.

While this may seem to be an undesirable side effect, it sometimes has optimization benefits. The top-to-bottom execution of the original code requires 10 cycles (for two iterations since the compiler unrolled the loop once), and execution of the remapped code requires 11 cycles (cycle 6 in the original code has been broken into two cycles at the basic block boundary),[1] but execution of the rescheduled code requires only seven cycles (Figure 6.23(c)). The cross-iteration dependences in this example dictate that only three cycles are needed between each original iteration (six for the compiler unrolled iteration), so this side-effect is not quite as aggressive as a compiler could be. By performing loop rotation, the top of the successive iteration is moved inline with the bottom of the previous iteration, enabling optimization and scheduling of iteration heads and tails. The new loop body effectively represents a software pipelined loop with continuous iteration overlap.

Future iterative layout and scheduling passes could, in theory, continue to improve the software pipelining effect. In the current example, operations from the top of the second iteration (bottom blue instructions in Figure 6.23(c)) are allowed to move into the bottom blocks of the previous iteration (top red blocks). A second pass would again rotate the new loop so that the

[1]The TGU could have placed the single first instruction from the block associated with B from cycle 6 in parallel with the two in cycle 7 speculating their parallelism. While the experimental EPIC would have stalled on a true dependence, other EPIC machines, such as Itanium, do not check intrablock dependences and may have produced illegal code.

top portion of the next iteration (the mixture of red and blue blocks) would be placed after the bottom of the next, allowing for further speculation (the original blue blocks would be cumulatively moved to the blue blocks of the previous iteration). This software pipelining side-effect process, however, is not as efficient as a compiler. Each pass of relayout and optimization only rotates the loop by one basic block and the rotation amount is limited to the arbitrary size of that block.

CHAPTER 7

RELATED WORK

The ROAR architecture represents multidisciplinary work merging the fields of compiler algorithms, run-time systems, and computer architecture. Hence, this work is derived from a number of concepts from each of these distinct fields as well as other multidisciplinary systems that also strive to provide run-time optimization capabilities. This chapter is comprised of three sections covering profiling, software-based optimization, and hardware-based optimization techniques. The first section relates to region selection (Chapter 4) while the second two related to region optimization (Chapters 5 and 6). Finally, a preliminary version of an optimization system taxonomy is provided in Appendix A that is intended to explore the breadth of choices relating to the aggressiveness and complexity of various systems.

7.1 Profiling

Control-flow profiles have been shown to provide invaluable information to an optimizer because they give insight into the frequently executed paths in the program. While profile-driven optimizations such as superblock formation and scheduling [47] have demonstrated significant improvements in performance, software vendors have been reluctant to add a profiling step to their compilation process. In some cases, compilers have employed static branch

prediction techniques to form estimated profiles with moderate success [51], [52]. More recently, a number of post-link-time optimization systems have been proposed that transparently and automatically profile and reoptimize programs. However, even when the cost of profiling is minimized, instrumentation-based profiling can still incur significant overhead, measured at between 11% and 424% slowdown [53].

Low-overhead methods of transparent profiling have been developed based on statistical sampling [54]-[57]. The most common approaches either periodically or randomly sample the program counter (PC) value and track the distribution of sampled PC values over the course of execution. The periodic form, also useful for statistical sampling of other processor events such as cache misses, is often implemented through a simple hardware counter. When the event occurs, a counter is incremented. Then, once the counter reaches a predetermined threshold, an interrupt is taken to the operating system where the current PC is reported and the counter is reinitialized. For PC sampling, the event is simply instruction execution. However, due to the latency between an event and the counter overflow interrupt and due to out-of-order execution, the PC reported during the interrupt is often several instructions away from the offending instruction. More recent implementations buffer the PC value of the offending instruction each time the counter is incremented so that proper PC value reporting can be accomplished. These approaches suffer from three primary drawbacks. First, the entire profile of each application must be continuously maintained at run time on the production system. Not only is there run-time overhead in collecting the numerous samples, there is additional overhead in performing software analysis on the collection. Second, the profile represents only average behavior over an extended period of time. And third, the latency of detecting variations in the program's

behavior can be great. The proposed Hot Spot Detector addresses these drawbacks by profiling only the most frequently executed instructions, those most likely to benefit from post-link optimization. The detector also thoroughly tracks instruction behavior over a short time window to provide an accurate and timely relative profile for each phase of execution.

Several more comprehensive profiling mechanisms have also been proposed. Basic Block Distribution Analysis [58] combines intense, periodic sample-based profiling to determine the composition of repetitive phases, but is not applicable to more general phasing behavior patterns. The *ProfileMe* [59] system introduced two primary improvements over sampling-based methods. First, the system provides a means for attributing a variety of events to specific instructions, whereas previous event counter mechanisms could only pinpoint offending instructions within a handful of cycles. The ability to correlate events to instructions is a key factor for performing microarchitecture-specific optimization. Second, the system allows for comprehensive random profiling of pairs of instructions in order to better understand their pipeline interaction. Like these systems, the Hot Spot Detector is able to correlate branch behavior with specific instructions, but it also has the ability to automatically filter out the unimportant branches. Furthermore, the Hot Spot Detector only requires servicing by either a hardware or software optimizer when a run-time optimization opportunity has been detected.

The Profile Buffer [60] is a hardware table that resides in the retirement stage of the microprocessor and consists of a small number of entries (in the range of 16-64) which are used to thoroughly track branch behavior. The operating system samples and resets the profile buffer when it becomes full, thus minimizing its run-time overhead, estimated at 2% - 5%. Over time, the samples are coalesced to form an edge weight profile. The dynamic working set signature

detector [61] is another hardware approach which intensively monitors the executing blocks for pattern shifts. This hardware component could be used to trigger the beginning of a new phase and determine whether or not the phase had already been detected, but generally has no mechanism for determining the actual instruction content of the phase.

7.2 Software-Based Dynamic Optimization

Run-time profiling and optimization of applications promises to deliver higher performance than is currently available with static techniques. A number of software- or firmware-based, dynamic optimization systems have emerged that optimize running applications, storing the optimized sequences into a portion of memory for extended execution. DAISY [62], FX!32 [63], UBQT [64], Aries [65], and more recently Transmeta [66] and BOA [67] are systems designed to perform dynamic code translation from one architecture to another. Early versions of DAISY and FX!32 were primarily concerned with providing architectural compatibility with subsequent enhancements targeting performance, while the Transmeta processors specifically utilize dynamic translation as a means for improving performance. Dynamo for HPPA [22], [68], a Dynamo derivative for x86 Windows [69], Wiggens/Redstone for Alpha [70], and Mojo for x86 Windows [71] are systems designed to improve application performance on the same architecture. Similarly, just-in-time compilers [72], [73] have been introduced to generate optimized code at run time from an intermediate representation such as Java bytecode. However, these systems often suffer from significant software overhead. Many of them utilize interpretation to comprehensively profile applications before generating optimized code sequences. Others

generate poorly optimized versions with embedded profiling counters to accomplish the same goal. However, time spent in an interpreter or spent executing probed code is overhead that must be reclaimed when executing the optimized code in order to see a performance benefit. These systems also require a software executive to monitor and control the reoptimization process. While ROAR uses a similar memory-based code storage technique, it uses a hardware structure that operates in parallel with the execution of the application for rapid profiling, analysis, optimization, and management.

Similar to Dynamo (the HPPA to HPPA software dynamic optimization system), the proposed Trace Generation Unit utilizes the real execution stream to select traces for optimization [74]. In Dynamo, instructions are initially interpreted while potential trace starting points are thoroughly profiled. Once an execution threshold for a starting point is reached, a trace is formed beginning at the starting point following the current execution stream. ROAR improves upon Dynamo's method by eliminating the overhead associated with interpretation and profiling by performing the profiling in hardware. Because the profiling overhead is low, all branches can be efficiently profiled and used to guide the trace formation process. As a software system, Dynamo provides flexibility because the trace formation and optimization choices can be customized and upgraded to address the needs of different usage environments. Furthermore, it can operate without any required support from the underlying hardware, and may be able to utilize a more global view of the code during optimization. However, the software overhead of Dynamo limits the overall benefit from any optimization performed. Additionally, if an application requires precise exceptions, optimization and any instruction reordering must be further restrained.

Most compilers improve fetch performance by reordering a program's blocks sequentially along expected frequent paths. Compilers often use profile information [47], [75] to organize the functions and blocks within functions, performing function inlining where appropriate. The Software Trace Cache [76] technique constructs traces at compile time that are similar to the traces constructed at run time by a trace cache (described in detail in Section 7.3). Spike, an Alpha architecture off-line link-time optimizer [77], [78], separates the hot blocks from the cold blocks using profiling information. Then, the hot blocks are reordered and formed into likely paths of execution. Finally, the hot blocks are analyzed and instructions that are only needed to support the cold paths are relocated into compensation code segments stored with the cold blocks. Spike operates on Alpha object files providing profile-guided cross-source-file optimizations during the linking process. Vulcan [79] from Microsoft and Lbx86 [80] from the IMPACT Research group are similar tools for binary optimization of x86 Windows applications. Unlike most of the static techniques described, ROAR also optimizes across library and dynamically linked library boundaries. This is a significant distinction because the software industry is migrating toward more modular component-based software.

By taking advantage of optimization opportunities chosen at compile time, dynamic compilation (DyC [81] and tcc [82]) has been used to perform optimized code generation at execution time. Run-time information, particularly the consistency of values, is used by the DyC software dynamic compiler to generate optimized code for regions which are annotated during compilation. The compiler sets up potential regions by generating template code for each opportunity with holes that can be filled by run-time detected values. For instance, a loop with run-time constant iteration count can be processed into a template that simply consists of the loop body

and space for the iteration count check. At run time, the iteration count can be detected, the loop potentially unrolled to a factor of the iteration count, and the iteration count check inserted. These templated regions can be selected automatically through use of code analysis and profile information [83]. While specific loops achieve impressive speedups (as much as 500%), whole-program generalization has yielded moderate improvements, especially in integer applications. By utilizing hardware support, a similar technique, called Compiler-directed Computation Reuse [84], takes advantage of run-time consistent values to eliminate entire regions of redundant computation through result memoization. However, these techniques rely upon regions chosen at compile-time and are limited by the effectiveness of the programmer or automated annotation mechanisms.

7.3 Hardware-Based Dynamic Optimization

Dynamic scheduling of instructions in hardware (out-of-order execution) has been used to improve instruction-level parallelism at run time [85], [86]. By reordering instructions at run time, execution times ideally can be reduced to the computation height of the code sequence. Dynamic height reduction relies on aggressive register renaming [87] to eliminate the output and antidependences that restrict out-of-order execution and code motion. Furthermore, the cost of long latency operations can be hidden by the concurrent execution of other instructions. However, the window from which instructions can be selected is typically small, and no record is kept of failed reordering attempts to prevent them in the future. In addition, since the hardware for this form of dynamic scheduling is located on the processor's critical path, there is

limited opportunity for more advanced optimizations. One goal of the ROAR system is to explore a processor architecture that only occasionally reoptimizes the code at run time instead of the constant reoptimization in out-of-order machines, similar to what the DIF microarchitecture provides [88]. DIF, dynamic instruction formating, collects instructions executed along frequent paths and dynamically bundles them into sets of independent parallel instructions. The sets are then formed into atomically executed groups and stored in a special hardware DIF cache. When a branch is mispredicted or scheduling assumption violated, processor state is restored to a checkpointed state taken at the beginning of the group. When grouping instructions, the scheduler analyzes the code for instruction-level parallelism and schedules the code to effectively utilize the available functional units. The ROAR architecture serves a similar purpose, but allows instructions to be stored in memory for long-term execution and provides rolling checkpointing of speculative values which enables arbitrarily long execution traces. Through occasional reoptimization, power consumption may be reduced through shorter optimizer hardware duty cycles, and pipeline complexity can be reduced by moving the scheduling hardware off of the critical path.

The Trace Generation Unit implements a set of run-time optimization techniques that attack a problem traditionally managed through either special-purpose hardware or compiler techniques. In order to achieve higher levels of instruction-level parallelism in superscalar machines, the fetch unit is required to supply multiple basic blocks per cycle to the execution units. Early designs include the sequential instruction caches and the collapsing buffer [89], which were designed to fetch several contiguous blocks across cache boundaries and nonsequential intracache line blocks, respectively, in a single cycle. However, the complexity of the

shift logic in the collapsing buffer may cause an undesirable increase in fetch latency. More recent solutions to the wide fetch problem involve reordering the code blocks into executed order and storing them into a special cache called the trace cache [90]. This cache organization has been shown to perform well, as instructions normally separated by taken branches can be fetched in a single cycle.

Optimizations beyond block reordering in a traditional trace cache have also been proposed [91], but these transformations have been limited to classical optimizations. Since all the instructions within a trace are likely to execute together, architectures where traces serve as the fundamental unit of work have been proposed. In the Trace Processor architecture [92], sequences of traces are predicted and dispatched as a unit. Since they are cached, fetched, and executed as a unit, they provide an opportunity for more aggressive optimization such as instruction rescheduling [93].

Because traces in the trace cache are short and often have brief lifetimes, the trace cache is a limited framework for dynamic optimization. One proposed system, called the rePLay framework [94], provides a microarchitecture in which instructions are collected into much longer traces and optimized by the hardware. In this system, an enhanced trace cache delivers units of execution called *frames* to the processor core. Each frame consists of a long sequence of instructions selected such that there is high probability of executing through to the end of the trace. Side-exit branches, all with a low exit probability, can then be converted into assertions, essentially making an assumption that the branch conditions will not be true. Thus, instruction reordering can be performed without regard to side-exits from the trace. A checkpoint of architectural register state is made prior to a frame's execution, and all stores in the trace

must be buffered until the end of the trace, since a frame must completely execute in order to commit any changes to register or memory state. A failed assertion test or exception anywhere in the trace signals that an assumption made during optimization has been broken, and that execution must be completely restarted using the original instructions from the instruction cache. By comparison, ROAR uses an explicit commitment mechanism that, upon an early trace exit due to a branch or an exception, can retain all of the nonspeculative (and committed speculative) work performed. This commitment mechanism, previously described as rolling commits, combined with a strict preservation of store ordering, also eliminates ROAR's need for a gated store buffer.

ROAR and rePLay exploit program execution variations at different granularities. ROAR forms new extracted hot spot regions (containing many traces) at intervals on the order of 10^5 to 10^7 cycles while rePLay detects a new frame every 10^2 cycles [20]. This difference allows rePLay to take advantage of short-lived trends in addition to stable patterns, but requires continuous detection and optimization of frames. ROAR focuses on phases whose useful lifetime is minimally hundreds of thousands of cycles so SHOP can amortize optimization costs and focus on traces likely to continue execution. Since ROAR utilizes a memory-based code cache, its transformations are persistent even across context switches. The ROAR architecture, as currently designed, detects and optimizes only a single hot spot at a time and may temporarily delay the formation of other hot spots until the current one is complete. Previous evaluation of rePLay has assumed a fully pipelined optimization mechanism. Using their default scheduling time of 1000 cycles per frame and an average frame detection rate of 100 cycles [20], this mechanism must be able to optimize an average of 10 frames in parallel throughout the

execution of an application. The ROAR framework fundamentally contains an automatic filtering mechanism designed to identify and optimize the truly important code regions.

Transmeta's Crusoe processors [66], [95] and IBM's BOA [67] also use a software system to translate, profile, and optimize programs at run time. Like the rePLay, they employ a shadow register file and gated store buffer to allow aggressive optimization of the translated code. In the presence of an exception, they roll back to a previous checkpoint and use software emulation on the the original code to provide precise exceptions. Similarly, IBM's DAISY [96] provides for precise exceptions using checkpoints. Upon an exception, the processor state is rolled back to the checkpoint. The checkpoint provides a base pointer into the original code, and a backwards examination of the translated instructions provides an offset to the original program address that should signal the exception.

Several recently proposed systems include a coprocessor in the hardware specifically designed to optimize instructions. One such mechanism is called the instruction path coprocessor (I-COP) [97]. This particular design is essentially a small load-store architecture with a small set of registers that is enhanced with several new instructions for pattern matching, bit manipulation, and data movement with the core processor's nonarchitected registers. While such coprocessors may provide a flexible optimization engine, they require their own instruction and data memory and functional units. We provide similar scheduler functionality through special-purpose hardware for reduced cost, though with potentially less flexibility. The SHOP, however, could be implemented by a mechanism similar to the I-COP.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Recent innovations in microprocessor design have given the processor more control over how to execute code optimally. ROAR techniques advance the state-of-the-art by allowing the processor to detect the most frequently executed code, to perform code straightening and partial function inlining, loop unrolling optimizations, and instruction rescheduling, and to deploy the code for immediate use, in a manner transparent to the user application. The Hot Spot Detector monitors the retired instruction stream, providing a relative profile of the most frequently executed instructions in the stream. Unlike other hardware profilers, the detector utilizes a continuous analysis algorithm to determine when a suitable region for run-time optimization is found. The Trace Generation Unit extracts traces from the instruction stream, filtering out infrequent paths by using the profiles stored in the detector. The generated trace sets are written into memory so that a ROAR processor may employ a traditional fetch mechanism. Further, the trace sets are passed through the Scheduler and Optimizer, which utilizes Precise Speculation to improve the instruction schedule to exploit available ILP. The detection and extraction of frequently executed code is done after the retirement stage of the processor, off the timing-critical paths.

Preliminary results show that the TGU optimizations applied by ROAR achieve significant fetch performance improvement at little extra hardware cost. For in-order core processors,

ROAR provides the ability to adapt to new input patterns and overcome module boundaries. ROAR's ability to identify hot spots early in their lifetime so that the vast majority of the execution of these hot spots is spent in optimized code is a key contributing factor to its success. In addition, because the generated code consists of important, persistent traces, ROAR creates opportunities for more aggressive optimizations.

ROAR presents a platform for pursuing a number of other promising dynamic optimization opportunities. First, the hot spot detection and trace filtering mechanisms could be applied to other trace-based dynamic optimization systems, hardware or software. For example, rePLay could use the BBB to filter out spurious traces in order to reduce optimizer bandwidth. Second, ROAR's hardware-centric approach could be exploited by other systems. Dynamo could employ the hardware features for fast profiling and possibly trace generation. Dynamic code generators and just-in-time compilers could use the scheduling hardware to quickly produce adequate instruction schedules instead of interpreting or performing high-overhead scheduling before code regions are even determined to be important.

Trace generation techniques can form code fragments across basic block boundaries where the compiler generally does not, such as across control-flow merge points, to produce higher performance code. In general, traces have superior local code performance when they match execution behavior, but may have very poor performance when they do not. When traces can be adapted to match execution behavior, a system can be created to automatically form customized traces that produce a sustained increase in performance. Compilers and region-based optimizers [98], however, consider a larger scope and have the potential to optimize beyond the bounds of code snippets and traces. However, they often encounter obstacles within the regions due to

control-flow merge points that limit scheduling and optimization (an instruction operand might have several different possible producers). Typically, these boundaries form code fragments that are shorter than traces which can reduce the benefits of instruction scheduling. Optimizations at the region granularity also typically require significant amounts of information to properly manage memory dependences and predicates that are currently not available to dynamic optimizers. ROAR, while designed as a platform for region-based optimization, currently uses trace-based optimizations because of the lack of analysis information. Future work will focus on the tools needed to retrieve and utilize deeper analysis information that will further improve the benefits provided by ROAR. This analysis information is likely to be generated and stored by the compiler. It should be noted that the compiler is a valuable optimization tool that should be co-designed with the dynamic optimizer to provide maximum synergy. Examination of ROAR-generated traces revealed that some traces had instruction schedules that closely matched the schedules originally made by the compiler. Some of the similarity stems from ROAR's restricted code motion due to memory dependences, but the compiler also successfully scheduled the instructions into tight POEs based upon machine resources. Furthermore, the compiler may set up delicate but fast code sequences, such as modulo scheduled loops, that could easily be slowed by an unaware dynamic optimizer. Clearly, both the compiler and dynamic optimizer have strengths that should both be exploited in a high-performance system.

Extension of ROAR ideas to managed code environments, such as .NET [7], could greatly improve dynamic compilation overhead. By utilizing the phasing concept, code regions that contain just the important functions could be generated and managed together on a per hot spot basis, much like ROAR's management system. This technique could provide compact,

high performance code that only requires generation-time analysis on the important functions instead of all touched functions. Furthermore, a generational hot spot code manager could be employed to retain the truly important hot spots while making room for newly detected regions. Exploitable program phases, however, do not often cover 100% of an application's execution. Rather, other less intensively executed periods exist that often glue the phases together. These regions may benefit from a less aggressive and smaller-scoped extraction and optimization mechanism. Therefore, a hierarchical approach may be ideal with a region-based approach, such as ROAR phase optimization, at the top followed by optimizations on individually collected traces below. Last, a more detailed study of the benefits provided by store sets and their representation in a ROAR-like environment is in order.

ROAR concepts could also lead to a new approach to profile-based off-line program optimization. Rather than using traditional aggregate or summarized execution profile weights, this approach would use the transparent Hot Spot Detector hardware to automatically detect execution phases and record branch profile information for each new phase. A code extraction algorithm [99] would then produce code packages, or regions, that are specially formed and partially inlined [100] for their corresponding phases [101]. The algorithm would rely on efficient information propagation and heuristic estimation techniques to compensate for the incomplete and often incoherent branch profile information that arises due to the nature of hardware profiling. The technique would avoid unnecessary code replication by focusing on hot code, making efficient connections between the original code and the new code, linking code regions at select points to facilitate phase transitions, and providing a platform for efficient optimization.

REFERENCES

[1] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 136–147.

[2] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, pp. 567–589, June 2001.

[3] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu, "A hardware mechanism for dynamic extraction and relayout of program hot spots," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 59–70.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 206–218.

[5] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.

[6] P. Robichaux, *Managing Windows 2000 Registry*. Sebastopol, CA: O'Reilly and Associates, 2000.

[7] D. S. Platt, *Introducing Microsoft .NET*. Redmond, WA: Microsoft Press, second ed., 2002.

[8] R. Ben-Natan and O. Sasson, *IBM Websphere Application Server: The Complete Reference*. New York, NY: Osborne McGraw-Hill, 2002.

[9] Hewlett-Packard Company, *Core Services Framework, Document Number CSF04DV10*. Palo Alto, CA, 2002.

[10] M. Schlansker and B. R. Rau, "EPIC: An architechure for instruction parallel processors," Hewlett-Packard Laboratory, 1501 Page Mill Road, Palo Alto, CA 94304, Tech. Rep. HPL-1999-111, February 2000.

[11] E. Nystrom, R. D. Barnes, M. C. Merten, and W. W. Hwu, "Code reordering and speculation support for dynamic optimization systems," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001, pp. 163–174.

[12] Standard Performance Evaluation Corporation, "Spec newsletter." http://www.spec.org.

[13] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*. Sebastopol, CA: O'Reilly and Associates, second ed., 1996.

[14] J. R. Larus, "Whole program paths," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999, pp. 259–269.

[15] W. W. Hwu, M. C. Merten, A. R. Trick, C. N. George, and J. C. Gyllenhaal, "Method and Apparatus for Instruction Execution Hot Spot Detection and Monitoring in a Data Processing Unit." U. S. Patent Application, University of Illinois at Urbana-Champaign, March 2000.

[16] B. Shriver and B. Smith, *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[17] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998, pp. 227–237.

[18] D. Bruening and E. Duesterwald, "Exploring optimal compilation unit shapes for an embedded just-in-time compiler," in *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000, pp. 13–20.

[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann Publishers, 1996.

[20] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel, and S. Lumetta, "Performance characterization of a hardware mechanism for dynamic optimization," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2001, pp. 16–27.

[21] S. J. Patel, T. Tung, S. Bose, and M. Crum, "Increasing the size of atomic instruction blocks by using control flow assertions," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000, pp. 303–316.

[22] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent dynamic optimization: The design and implementation of dynamo," Hewlett-Packard Laboratories Cambridge, Tech. Rep. HPL-1999-78, June 1999.

[23] D. Carmean, "The pentium 4 processor." presented at Hot Chips 13, Stanford University, Palo Alto, CA, August 2001.

[24] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*, vol. 48, pp. 193–204, February 1999.

[25] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference, Document Number 243191*. Santa Clara, CA, 1997.

[26] Compaq Computer Corporation, *Alpha Architecture Handbook: Version 4, Order Number EC-QD2KC-TE*. Compaq Computer Corporation, 1998.

[27] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Volume 3: Instruction Set Reference, Document Number 245319-003*, December 2001.

[28] S. McFarling, "Combining branch predictors," Digital, WRL, Tech. Rep. TN-36, June 1993.

[29] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001, pp. 15–25.

[30] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, November 1992, pp. 808–817.

[31] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000, pp. 57–68.

[32] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998, pp. 142–153.

[33] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *ACM Transactions on Computer Systems*, vol. 11, November 1993.

[34] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 36–44.

[35] E. M. Nystrom, "Architectural support for persistent, dynamic code transformations," M.S. thesis, University of Illinois, Urbana, IL, 2002.

[36] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind, "Optimizations and oracle parallelism with dynamic translation," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, November 1999, pp. 284–295.

[37] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallet, "Local microcode compaction techniques," *ACM Computing Surveys*, vol. 12, pp. 261–294, September 1980.

[38] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[39] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 330–335.

[40] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 266–275.

[41] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1995, December 1995.

[42] W. W. Hwu, J. W. Sias, M. C. Merten, E. M. Nystrom, R. D. Barnes, C. J. Shannon, S. Ryoo, and J. V. Olivier, "Itanium performance insights." presented at Microprocessor Forum, San Jose, CA, October 2001.

[43] C. J. Shannon, "The IMPACT SC140 code generator," M.S. thesis, University of Illinois, Urbana, IL, 2002.

[44] AJ. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research," in *Proceedings of the Workshop on Workload Characterization*, September 2000, pp. 83–100.

[45] AJ. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC 2000 benchmark workload for simulation-based computer architecture research," *Computer Architecture Letters*, vol. Volume 1, May 2002.

[46] J. W. Sias, W. W. Hwu, and D. I. August, "Accurate and efficient predicate analysis with binary decision diagrams," in *Proceedings of 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 112–123.

[47] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[48] A. Nicolau, "Run-time disambiguation: Coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[49] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[50] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[51] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313.

[52] B. L. Deitrich, B. C. Cheng, and W. W. Hwu, "Improving static branch prediction in a compiler," in *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, October 1998, pp. 214–221.

[53] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1319–1360, July 1994.

[54] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" in *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, October 1997, pp. 1–14.

[55] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," in *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, October 1997, pp. 15–26.

[56] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 85–96.

[57] Intel Corporation, *Intel Itanium Processor Reference Manual for Software Development, Document Number 245320-003*, December 2001.

[58] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001, pp. 3–14.

[59] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 292–302.

[60] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 36–45.

[61] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 233–244.

[62] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 26–37.

[63] R. J. Hookway and M. A. Herdeg, "Digital FX!32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, pp. 3–12, August 1997.

[64] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable binary translation at low cost," *IEEE Computer*, pp. 60–66, March 2000.

[65] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent execution, no recompilation," *IEEE Computer*, pp. 47–52, March 2000.

[66] A. Klaiber, "The technology behind Crusoe^{TM} processors," Transmeta Corporation, http://www.transmeta.com, tech. rep., January 2000.

[67] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *IEEE Computer*, pp. 54–59, March 2000.

[68] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000, pp. 1–12.

[69] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dyanmic optimization framework for windows," in *Proceedings of the Fourth ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[70] D. Dever, R. Gorton, and N. Rubin, "Wiggens/Redstone: An on-line program specializer." presented at Hot Chips 11, Stanford University, Palo Alto, CA, August 1999.

[71] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gilles, "Mojo: A dynamic optimziation system," in *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[72] A.-R. Adl-Tabatabi, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, effective code generation in a just-in-time java compiler," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 280–290.

[73] Sun Microsystems, "The Java HotSpot^{TM} virtual machine," Sun Microsystems, http://java.sun.com, tech. rep., 2001.

[74] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, December 2000, pp. 202–211.

[75] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[76] A. Ramirez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *Proceedings 1999 International Conf. on Supercomputing*, June 1999, pp. 119–126.

[77] R. S. Cohn and P. G. Lowney, "Hot cold optimization of large Windows/NT applications," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 80–89.

[78] R. S. Cohn, D. W. Goodwin, and P. G. Lowney, "Optimizing Alpha executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, no. 4, pp. 3–19, 1997.

[79] A. Srivastava, "Vulcan," Microsoft Research, Tech. Rep. TR-99-76, September 1999.

[80] M. C. Merten, "A framework for profile-driven optimization in the IMPACT binary reoptimization system," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1999.

[81] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers, "Annotation-directed run-time specialization in C," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, June 1997, pp. 163–178.

[82] M. Poletto, D. Engler, and M. Kaashoek, "tcc: A system for fast, flexible, and high-level dynamic code generation," in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 109–121.

[83] M. Mock, C. Chambers, and S. J. Eggers, "Calpa: A tool for automating selective dynamic compilation," in *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000, pp. 291–302.

[84] D. A. Connors and W. W. Hwu, "Compiler-directed computation reuse: Rationale and initial results," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999, pp. 158–169.

[85] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[86] W. W. Hwu and Y. Patt, "Checkpoint repair for high performance out-of-order execution machines," *IEEE Transaction on Computers*, vol. C-36, pp. 1496–1514, December 1987.

[87] R. M. Keller, "Look-ahead processors," *ACM Computing Surveys*, vol. 7, pp. 177–195, December 1975.

[88] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 13–25.

[89] T. M. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 333–344.

[90] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 24–34.

[91] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings 31st Annual IEEE/ACM International Symposium on Microarchitecture*, December 1998, pp. 173–181.

[92] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997, pp. 138–148.

[93] Q. Jacobson and J. E. Smith, "Instruction pre-processing in trace processors," in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999, pp. 125–129.

[94] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, June 2001.

[95] M. J. Wing and G. P. D'Souza, "Gated Store Buffer for an Advanced Microprocessor." U. S. Patent No. 6,011,908, Transmeta Corporation, January 2000.

[96] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and compilation," *IEEE Transactions on Computers*, vol. 50, pp. 529–548, June 2001.

[97] Y. Chou and J. P. Shen, "Instruction path coprocessors," in *Proceedings 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 270–281.

[98] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 158–168.

[99] R. D. Barnes, "Extracting hardware-detected program phases for post-link optimization," M.S. thesis, University of Illinois, Urbana, IL, 2002.

[100] T. Way and L. L. Pollock, "A region-based partial inlining algorithm for an ILP optimizing compiler," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002, pp. 552–556.

[101] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu, "Vacuum packing: Extracting hardware-detected program phases for post-link optimization," to appear in *Proceedings of 35th Annual International Symposium on Microarchitecture*, November 2002.

APPENDIX A

TAXONOMY

In order to better compare the ROAR architecture with other optimization systems, a rough taxonomy of such systems is presented in this appendix. This taxonomy appears here in preliminary form as developed by Chris Newburn of Intel Corporation, in conjunction with Matthew Merten, Ronald Barnes, and Wen-mei Hwu of the University of Illinois. In each of the categories within the taxonomy, the characteristics that best describe ROAR are italicized. Where appropriate, an annotation may appear recognizing a particular system that utilizes a more unusual strategy. Of course, subtle flavors of any particular strategy may exist that could perhaps also warrant further delineations, but attempts have been made to cover all of the major options. For example, on-demand occasional reoptimization could be triggered by a variety of conditions including a new trace formation, a new hot spot detection, a change in a semiconstant value, etc. Likewise, a comprehensive system may use more than one strategy in any particular category. This appendix is designed to help delineate the possibilities of program optimization but is not intended to argue their merits. Most strategies have at least some merits when comparing aggressiveness to simplicity of implementation.

I. Dynamism

- pure static
- once, with feedback (e.g., profile-guided compilers)
- each time there is a recompile
- each time a module is linked/loaded/run
- each time a previously untouched code region or page is touched (e.g., DAISY)
- fixed period (e.g., after every context switch)
- each time a new execution characteristic is identified (e.g., phase change, new hot spot in ROAR, new trace in rePLay)
- continuous (e.g., out-of-order execution)

II. When to compute info needed to perform optimization, and similarly when to actually perform optimization

- module compile - during formation of .obj
- link - during formation of .exe or .dll
- predistribution - bring in modules from expected dlls (e.g., Vulcan)
- install - when .exe and .dll installed on machine
- load - bring all linked modules together
- run - during execution
  - invocation of optimization
    - first execution (e.g., DyC, JITs)
    - *at phase changes*
    - fixed period
    - after passing hot threshold
    - stall-time (e.g., on Icache miss)
    - idle-time - application is idle for lack of input, etc.
    - continuously (e.g., O-o-O)
  - perform optimization
    - cycle-by-cycle (e.g., O-o-O)
    - parallel process on separate processor on SMT
    - *dedicated hardware components*
- between runs
  - between every run
  - when profile data change exceeds threshold of interest

III. What info needed to optimize

- performance problems
  - branch mispredictions
  - cache misses
  - problematic code sequences
  - resource conflicts
  - assumptions on which a HW or code-generation policy is based on are violated
- specialization
  - value profiles
    - size of value set
    - phasing behavior
    - predictability
      - counters
      - context, correlation with path info
      - context, correlation with other values
    - *pointer aliases*
  - *branch profiles*
  - dataflow
  - other program analyses
- thread-level parallelism
  - dependences
  - value profiles (see above)
  - latencies
  - resource utilization

IV. How to profile

- static analysis - statically generated profile estimates (e.g., Ball and Larus)
  - within function
  - within module
  - whole program
  - arbitrary region
- binary instrumentation
  - same regions as static analysis
- source instrumentation
  - same regions as static analysis

- via interpretation (e.g., Dynamo, iSpike)
- *via compiled emulation* (e.g., many compilers during the initial compilation)
- via generation of application with internal probes (e.g., many compilers)
- performance monitors, mechanisms (see where to profile)
  - characterization by counting
    - *short-term - profile snapshots*
    - long-term - averages
    - whole program - actual profiles
  - characterization by tracing
    - branch tracing (e.g., Pentium4 Processor)
  - characterization by sampling
    - location imprecise (e.g., Pentium Pro Processor)
    - precise (e.g., ProfileMe, Pentium4 Processor)

V. Profiling complexity

- *simple event*
- qualification
- counts per cycle above a threshold (e.g., Pentium4 Processor)
- code range (e.g., Itanium Processor)
- data address range (e.g., Itanium Processor)
- opcode mask (e.g., Itanium Processor)
- thread (e.g., Pentium4 Processor)
- privilege level (e.g., Itanium Processor, Pentium4 Processor)
- edge detection (e.g., Pentium4 Processor)
- branch type (e.g., Itanium Processor)
- speculation (e.g., Pentium4 Processor)
- relational
  - cascading (e.g., Pentium4 Processor)
  - prioritized stall causes (e.g., Itanium Processor)
  - delay between events (e.g., Pentium4 Processor)
  - delay above a threshold (e.g., Pentium4 Processor)
  - 2 instructions (e.g., ProfileMe)
- comprehensive - all events (e.g., ProfileMe)

VI. Where to profile

- location
  - everywhere
  - *regions*
  - random
  - selected IP ranges
    - hot code
  - selected IPs
    - on critical path
    - where optimization can be gainfully applied
  - problematic events
    - microarchitecture idiosyncrasy (e.g., absent bypass path on Itanium Processor)
    - other known performance problems
    - characterization indicates above interesting threshold
  - *select classes of instructions* (e.g., branches)
- who decides where to profile
  - fixed
    - selected cases, but everywhere (e.g., branches or loads)
  - directed to interesting IPs/IP ranges/events
    - SW analyzes
    - HW finds/analyzes
      - random
      - temporary fixed ranges
      - fixed algorithm
      - *adaptive/reconfigurable algorithm*
      - under direction from SW: programming fixed set of events to count/profile
      - under direction from SW: able to store algorithm

VII. Program Phasing/Locality

- def: some variable is reasonably constant for a period of time
- variables
  - *working sets and working set sizes*
  - data cache lines
  - traces
  - *branch behavior*
  - dependence info

  - alias info
  - values
- predictability
  - phase sequence
  - *phase length*
  - granularity
    * from run to run
    * within a run
- persistence
  - short phases - handfuls of cycles
  - *medium phases - millions of cycles*
  - long phases - seconds of cycles
  - across runs
- detection/training time
  - *short - suitable for HW*
  - long - need to predict, may need SW (e.g., memory producer and consumer)
- detection/training time relative to persistence
- end of a phase / new phase
  - *tolerance to minor variations*

VIII. HW/SW interaction for profile info collection and usage

- SW tells HW where/how to look/what to look for
  - profile these critical dependences to enable coarse-grained code motion or spur speculative threading
  - could gain much from specialization based on var's value set
  - configure hardware to check assertions
- HW profiles to catch hot spot/important events/dependences
  - *hot spots*
  - problem loads and branches
  - alias frequency
  - value profile
- SW does more analysis/optimization
  - identify critical path leading to problem load/branch, trigger point
  - specialize based on value profile
  - coarse-grained speculative code motion/multiversion code

- HW does more analysis/optimization
  - build dep graph, create data-driven thread
  - specialization
  - short circuiting
  - memoization

APPENDIX B

HOT SPOT DETECTION RESULTS

Figures B.1-B.8 present detailed hot spot detection information for the applications presented in Figure 4.3. For each application, the static instruction count and post-detection percentage of the dynamic instructions for each hot spot are depicted.

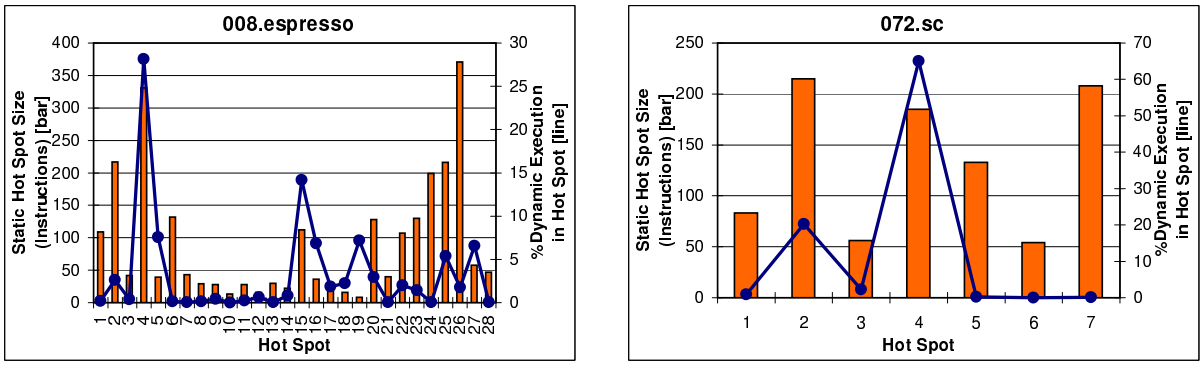


Figure B.1 Hot spot detection results for *008.espresso* and *072.sc*.

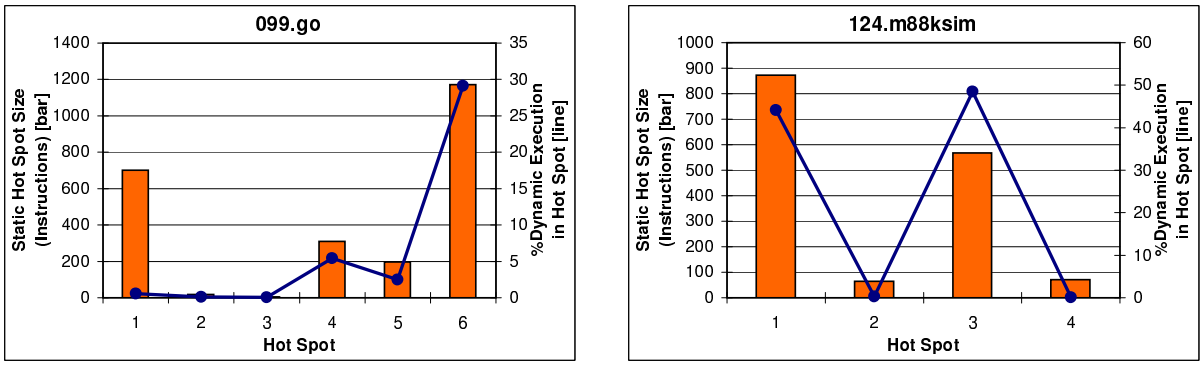


Figure B.2 Hot spot detection results for *099.go* and *124.m88ksim*.

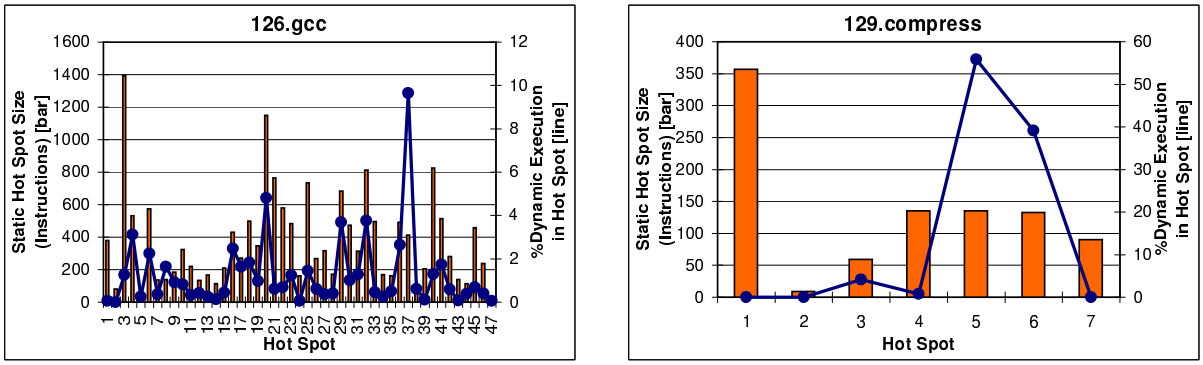


Figure B.3 Hot spot detection results for *126.gcc* and *129.compress*.

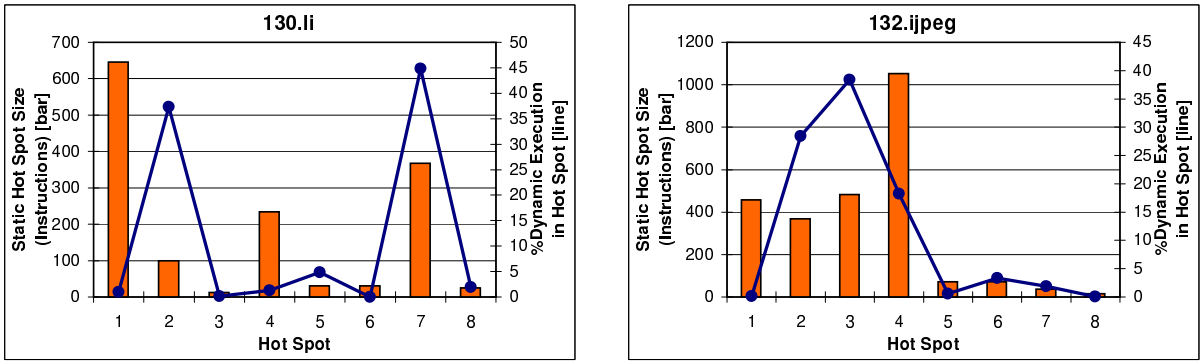


Figure B.4 Hot spot detection results for *130.li* and *132.ijpeg*.

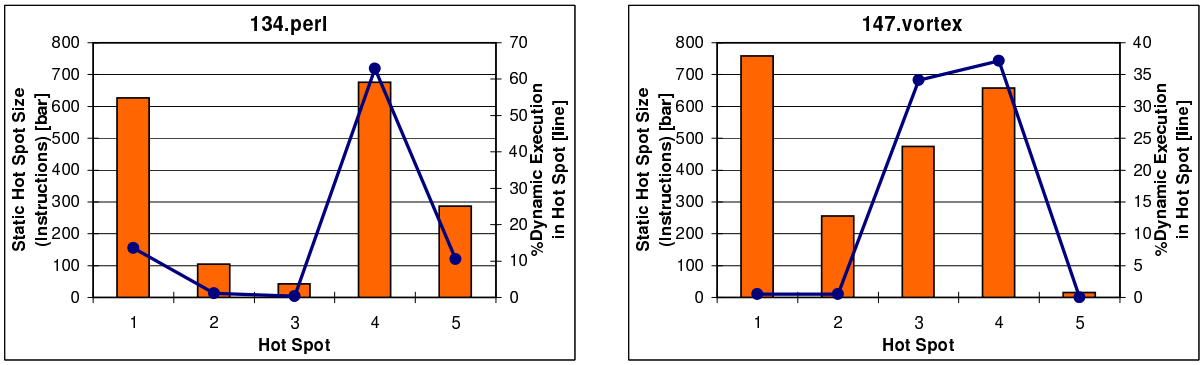


Figure B.5 Hot spot detection results for *134.perl* and *147.vortex*.

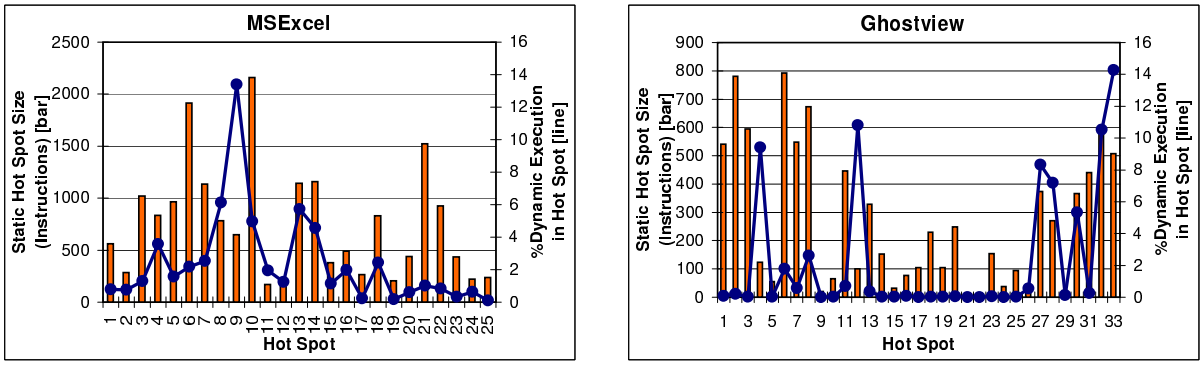


Figure B.6 Hot spot detection results for *MSExcel* and *Ghostview*.

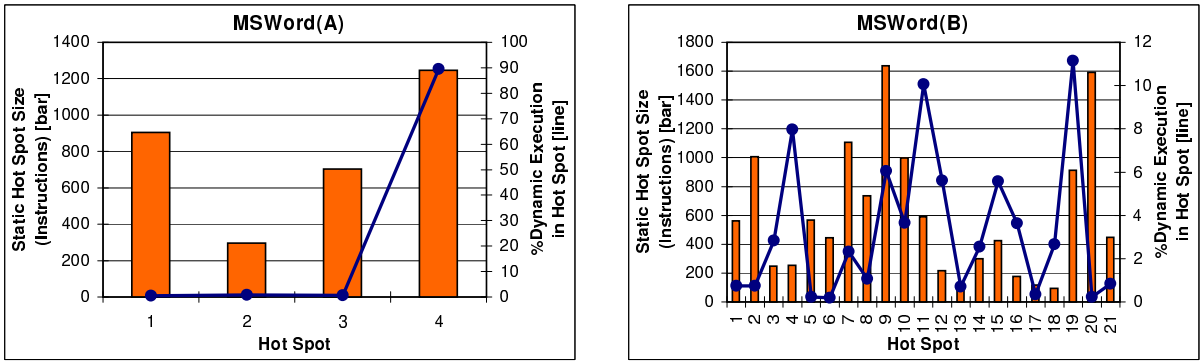


Figure B.7 Hot spot detection results for *MSWord(A)* and *MSWord(B)*.

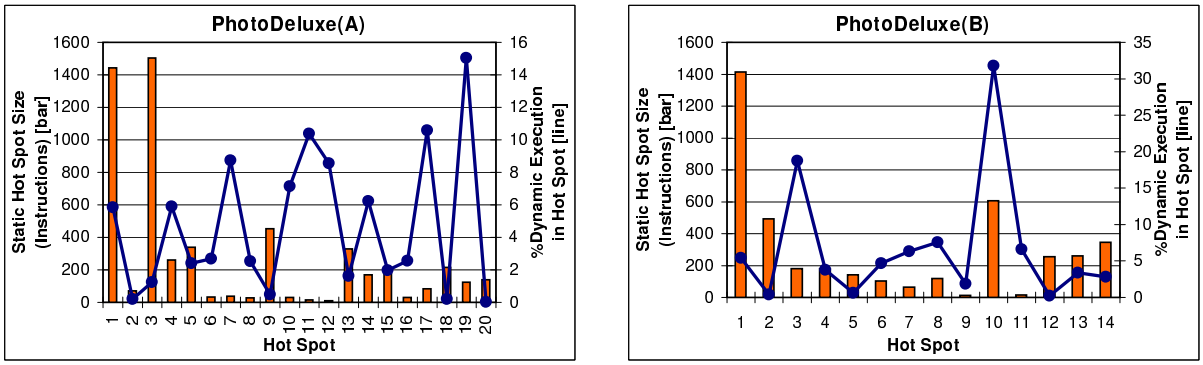


Figure B.8 Hot spot detection results for *PhotoDeluxe(A)* and *PhotoDeluxe(B)*.

VITA

Matthew Carl Merten was born on May 3, 1974, in St. Paul, Minnesota. He grew up in Shoreview, Minnesota, where he graduated from Mounds View High School in 1992. A portion of his senior year was spent at neighboring Bethel College through Minnesota's Post Secondary Enrollment Option for high school students. Matthew earned his Bachelor of Science degree in Computer Engineering with Highest Honors from the University of Illinois at Urbana-Champaign in May of 1996. He continued at the University of Illinois where he earned his Master of Science degree in Electrical Engineering in June of 1999 and his Doctor of Philosophy degree in Electrical Engineering in August of 2002. Matthew studied compiler construction, dynamic optimization, binary optimization, and modulo scheduling as a Graduate Research Assistant with the IMPACT research group under the direction of Professor Wen-mei Hwu. Matthew's appointment was in conjunction with the Center for Reliable and High Performance Computing within the Coordinated Science Laboratory. He has spent summers working with Unisys Corporation in Roseville, Minnesota, Hewlett-Packard Corporation in Cupertino, California, and IMPACT Technologies, Incorporated, in Champaign, Illinois, and has consulted for Microsoft Corporation, in Redmond, Washington. After completion of his Ph.D. degree, Matthew will join Intel Corporation in Hillsboro, Oregon, as a microprocessor architect.