

ABSTRACT

HAZELWOOD, KIM MICHELLE

Dynamic Optimization Infrastructure and Algorithms for IA-64

(Under the direction of Professor Thomas M. Conte)

Dynamic optimization refers to any program optimization performed after the initial static compile time. While typically not designed as a replacement for static optimization, dynamic optimization is a complementary optimization opportunity that leverages a vast amount of information that is not available until runtime. Dynamic optimization opens the doors for machine and user-specific optimizations without the need for original source code.

This thesis includes three contributions to the field of dynamic optimization. The first main goal is the survey of several current approaches to dynamic optimization, as well as its related topics of dynamic compilation, the postponement of some or all of compilation until runtime, and dynamic translation, the translation of an executable from one instruction-set architecture (ISA) to another.

The second major goal of this thesis is the proposal of a new infrastructure for dynamic optimization in EPIC architectures. Several salient features of the EPIC ISA prove it to be not only a good candidate for dynamic optimization, but such optimizations are essential for scalability that is up to par with superscalar processors. By extending many of the existing approaches to dynamic optimization to allow for offline optimization, a new dynamic optimization system is proposed for EPIC architectures. For compatibility reasons, this new system is almost entirely a software-based solution, yet it utilizes the hardware-based profiling counters planned for future EPIC processors.

Finally, the third contribution of this thesis is the introduction of several original optimization algorithms, which are specifically designed for implementation in a dynamic optimization infrastructure. Dynamic if-conversion is a lightweight runtime algorithm that converts control dependencies to data dependencies and vice versa at runtime, based on branch misprediction rates, that achieves a speedup of up to 17% for the SpecInt95 benchmarks. Several other algorithms, such as predicate profiling, predicate promotion and false predicate path collapse are designed to aid in offline instruction rescheduling.

Dynamic Optimization Infrastructure and Algorithms for IA-64

by

Kim Michelle Hazelwood

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh

2000

Approved By:

Dr. Thomas M. Conte, Chair of Advisory Committee

Dr. Eric Rotenberg

Dr. Injong Rhee

Biography

Kim Hazelwood was born on July 23, 1976 in Richmond, Virginia and grew up in Melbourne, Florida. She graduated first in her class with a Bachelor of Science degree in Computer Engineering from Clemson University in August 1998. During her years as an undergraduate, she participated in the cooperative education program working with Kemet Electronics Corporation in Greenville, South Carolina. She also participated in the Calhoun College Honors Program at Clemson University, as well as the Summer Undergraduate Research Program sponsored by the National Science Foundation where she worked in the Parallel Architecture Research Laboratory.

She then went on to the master's program at North Carolina State University working under the direction of Professor Thomas M. Conte. While working toward her master's degree, Kim spent a summer interning with the CarbonFIRE Dynamic Optimization group at Hewlett-Packard Company in Cupertino, California, and a summer with the Dynamo group at Hewlett-Packard Laboratories in Cambridge, Massachusetts.

She plans to continue her graduate education working toward her Ph.D. in Computer Engineering.

Contents

List of Figures	v
List of Tables	vii
1 Introduction	1
1.1 Motivation for Dynamic Optimization	1
1.2 Current Approaches	3
1.3 Dynamic Compilation vs. Optimization	4
1.4 Dynamic Translation vs. Optimization	5
1.5 Layout of the Thesis	5
2 Background	6
2.1 Dynamic Optimization Efforts	6
2.1.1 Dynamo - HP Labs	6
2.1.2 Morph	8
2.1.3 Evolutionary Compilation	10
2.2 Dynamic Compilation Efforts	11
2.2.1 DyC	11
2.2.2 DCG	13
2.2.3 Dynamo - Indiana University	13
2.2.4 Runtime Optimization	14
2.2.5 Tempo	16
2.2.6 Summary	17
2.3 Translation and Emulation Efforts	17
2.3.1 DAISY	17
2.3.2 Crusoe	18
2.3.3 FX!32	19
2.4 Profiling Monitors For Feedback-Directed Optimization	19
2.4.1 ProfileMe	20
2.4.2 Profiling for Hot Spots	21
2.4.3 The Profile Buffer	22
2.4.4 Itanium Monitors	22
2.4.5 Summary	23
2.5 Useful Tools for Dynamic Optimization	23

2.5.1	DynInstAPI	23
2.5.2	EEL	24
2.5.3	OM	24
3	Salient Features of EPIC Architectures	25
3.1	New Features of IA-64	25
3.1.1	Large Register File	27
3.1.2	IA-64 Instructions	27
3.1.3	Explicit Parallelism	27
3.1.4	Memory Access and Speculation	27
3.1.5	Control Speculation	28
3.1.6	Data Speculation	28
3.1.7	Predication	29
3.1.8	Stacked Registers (Register Windows)	29
3.1.9	Software Pipelining Support	30
3.1.10	Register Stack Engine	30
3.2	Summary	31
4	DO: A Dynamic Optimization Infrastructure	32
4.1	An Extension of Dynamo	33
4.2	Performance Improvements Achieved by HPL's Dynamo	34
4.3	A High-Level Description	35
5	Dynamic Optimization Opportunities and Algorithms	38
5.1	Sampling Study: Branch Mispredictions	38
5.1.1	Misprediction Sampling Based on First N Occurrences	38
5.1.2	Adaptive Warmup Exclusion	40
5.2	New Dynamic Optimization Algorithms	42
5.2.1	Dynamic If-Conversion	43
5.2.2	Dynamic Reverse If-Conversion	48
5.2.3	Dynamic Rescheduling for Scalability	50
5.2.4	Predicate Promotion with Double Predication	51
5.2.5	False Predicate Path Collapse	52
5.2.6	Predicate Profiling and Prediction	52
5.2.7	Dynamic Instruction Hint Adjustment	53
6	Conclusions and Future Work	55
	Bibliography	57

List of Figures

2.1	How HPL's Dynamo Works [6].	7
2.2	The Harvard University Morph Project [41].	9
2.3	Sathaye's Evolutionary Compilation System [36].	11
2.4	UW's DyC Dynamic Compilation System [18].	12
2.5	Kistler's Dynamic Runtime Optimization System [25].	15
2.6	Illinois' Monitor Table Hardware for Hot-Spot Detection.	21
2.7	The Profile Buffer Hardware [10].	22
2.8	Runtime Code Insertion via DynInstAPI.	24
3.1	An IA-64 Instruction Bundle.	26
3.2	An IA-64 Instruction Sequence without Control Speculative Loads.	28
3.3	An IA-64 Instruction Sequence with Control Speculative Loads.	28
3.4	An IA-64 Instruction Sequence without Data Speculative Loads.	29
3.5	An IA-64 Instruction Sequence with Data Speculative Loads.	29
3.6	Sample C Code with Branches.	30
3.7	Corresponding IA-64 Predicated Instructions.	30
4.1	Dynamo Overhead as a Percentage of Execution Time.	33
4.2	The Smallest Number of Fragments that Account for 90% of the Total Execution Time.	34
4.3	Wall Clock Time Comparison of Dynamo Performance to Native Execution.	35
4.4	The DO Dynamic Optimization Infrastructure.	36
5.1	The Average Difference between Sample (First N Branch Occurrences) and Actual Misprediction Rates for all Branches in the SPECint95 Benchmarks.	39
5.2	Error Values for a 99% Confidence Interval.	39
5.3	Misprediction Rate Over Time for Certain Hard-to-Predict Branches in SPECint95.	40
5.4	Determining the End-of-Warmup Condition.	41
5.5	Mean Difference between Sample (First N Branch Occurrences Ignoring Warm-up Period) and Actual Misprediction Rates.	41
5.6	Effect of Varying the Warm-up Completion Threshold Value.	42
5.7	If-Conversion Example.	44
5.8	A Predication Algorithm.	45
5.9	A General Predication Algorithm.	45
5.10	Dynamic If-Conversion Algorithm.	46

5.11	Maximum Allowable Branch Distance.	46
5.12	Speedup from Applying the Dynamic If-Conversion Algorithm.	47
5.13	Number of Branches Converted to Predicates.	47
5.14	Percent of Branch Misprediction Eliminated using Dynamic If-Conversion.	48
5.15	The Dynamic Reverse If-Conversion Algorithm.	49
5.16	Speedup Resulting from Dynamic Reverse If-Conversion.	49
5.17	Actual Number of Predicate Traces Converted to Branches.	50
5.18	Dynamic Rescheduling.	51
5.19	IA-64 Instructions with an Instruction Group Boundary.	51
5.20	Breaking the IA-64 Instruction Group Boundary.	51
5.21	An Example of False Predicate Path Collapse.	52
5.22	Predicate Profiling Mechanism.	53

List of Tables

2.1	Summary of past approaches.	17
3.1	The six IA-64 instruction types.	26
3.2	Possible instruction combinations.	26
5.1	IA-64 branch instruction completers.	53

Chapter 1

Introduction

When inspecting the basic run-time performance equation, $\text{runtime} = \text{instruction count} * \text{cycles per instruction} * \text{cycle time}$ [21], it becomes clear that a reduction in the overall runtime of a program can be accomplished by reducing any of the three variables - instruction count, cycles per instruction, or cycle time. The ideas discussed in this thesis attempt to reduce the run time of a given program by reducing both its instruction count and its cycles per instruction. This run-time reduction is not the result of new compiler techniques, as is typically the case. A dynamic optimization system is presented, which supplements the optimization performed by the static compiler and allows for a uniform level of optimization for all executables.

Dynamic Optimization refers to any optimization of software that is performed after the initial compile. Dynamic optimizations can be performed as the program is running or anytime thereafter, such as during the exit procedure of a program or even after the program has exited - during the idle time of the computer. Dynamic optimization can be used to leverage a large set of run-time information that is not available to the static compiler. This information can range from run-time constants to operator usage patterns. Such information can provide the foundation for an entirely new set of optimization algorithms.

Dynamic optimization is especially challenging due to the strict performance requirements for runtime optimization. Typically, our industry is not willing to accept a new idea unless performance improvements are seen immediately. Therefore, the standard lower-bound for dynamic optimization performance is that it must result in an overall system that performs at least as well as a system that does not employ dynamic optimization. While it is possible that users will permit some program slowdown if it is a one-time occurrence, this is only true in the case that performance of the program is improved to the extent that the average runtime over time is notably improved.

1.1 Motivation for Dynamic Optimization

Several factors may lead to the decision to implement a dynamic optimization system. Often, decisions made by a static compiler are not well suited for the runtime behavior of the program. This may be due a variance in the usage patterns of the client, a change or upgrade of hardware (such as the pipeline structure) since the initial compile

time, or simply an overly aggressive static compilation decision. The current software development cycle does not provide an opportunity for correcting or changing decisions made by the static compiler. Dynamic optimization can provide this flexibility in the software development cycle.

The limitations of a static-only optimization scheme make dynamic optimization an appealing option. Runtime values and branch behavior are known and can be leveraged in an effective manner. Some of the most prominent observations that led up to the need for dynamic optimization are described in the next sections.

Software Vendor Optimization Levels. One justification for a dynamic optimization system concerns the large amounts of off-the-shelf software packages that are purchased every year. Ideally, all software vendors would employ the highest level of optimization during their compilation process. Yet this is often not the case. Vendors may forgo high optimization levels because of the compile time necessary or because of the difficulty of debugging an executable with a high level of optimization. The varying level of compile technology available in the compilers used by software vendors has resulted in an unknown level of optimization in off-the-shelf software products and consumers have no way of ensuring that high levels of optimization were employed in the products they purchase. Dynamic optimization allows even the latest advances in optimization technology to be introduced to existing software, therefore ensuring optimal performance. For instance, Hewlett-Packard Laboratories has shown that programs originally compiled with level-2 optimization (-O2) running under their Dynamo dynamic optimization system performs as well as the same program compiled with level-4 optimization (-O4) without dynamic optimization [6].

Processor Changes Within an ISA. Yet another reason for choosing to implement dynamic optimization concerns processor upgrades within a processor family and cross-generation compatibility between processor families [11, 13, 36]. A program compiled for a given instruction-set architecture (ISA) can be run on any processor implementing that ISA. The program is only optimized, however, for execution on a processor with an identical execution pipeline, identical function units, and identical instruction availability. Variances in any of these attributes result in a program that is not employing the available functionality of the processor. Even worse, incorrect execution is possible for processors that do not check instruction dependencies at runtime, such as traditional VLIW processors or RISC processors that have load-use delay slots.

Allowing a user to recompile for his/her own system could easily provide for custom optimization, yet in the current software development cycle a recompilation requires original source code. Clearly, software vendors would not want to release their source code due to the proprietary information available within. Releasing versions of the executable for every single processor would furthermore be quite tedious. Dynamic optimization enables custom optimization for any processor within an ISA from a single executable, without the need for original source code.

Changes in Usage Patterns. Ideally, a program should be optimized based on how it is used. Unfortunately, usage patterns vary from person to person, and even one person's usage may vary over time [36] as they discover new features of the software. Since conventional compilation optimizes the program only once and before the user runs the program, optimization will either (1) not take into account actual user behavior, or (2) base optimiza-

tion on an average profile. There is no guarantee that the average profile information is at all representative of typical user behavior. Representative profile information is difficult to produce because of the large variance among users, and over time for the same user. This sheds doubt on the effectiveness of a unified set of profile data.

Dynamic optimization allows for software that is optimized for the particular user. Code can be optimized to better represent actual runtime behavior for each and every run of a program. It allows for continued optimization as the client's usage patterns change. This level of customization is simply not possible with conventional static optimization methods.

Trend Toward Static Scheduling. While most of the processors of the past rely on dynamic scheduling of instructions, an upcoming generation of processors contains a simpler processor core that relies on static scheduling. Scheduling programs once, and even before the program executes, increases the importance of accurate profile data. Inaccurate profiles result in poor scheduling decisions that will affect every run of a program [36]. Because static scheduling involves a great deal of speculation, poor scheduling choices can greatly harm runtime performance. At this time, unfortunately, there is no widely accepted and used means for correcting poor scheduling choices by the static scheduler. Dynamic optimization, on the other hand, may be effectively used to leverage runtime information in order to recognize poor static scheduling choices and correct them.

Power Reduction. A well-designed dynamic optimization system has great potential for venturing into other domains, such as power reduction. There are many opportunities for noting the runtime power consumption and making changes that focus on consuming less power. For example, code transformations can be performed that result in less bit flips between sequences of instructions [39]. Reduced bit flips result in reduced power consumption. Rather than having a NOP instruction equate to adding two zeros together, it could add the inputs of the previous instruction while storing the result in a free register. As processors become more and more power hungry, dynamic optimization could become an essential tool for balancing power and performance in a case-by-case basis.

Complementary Optimization Opportunity. There is an abundance of information unavailable at static compile time that can be leveraged by dynamic optimizations. Optimizations can be based on run-time constants and may span library and function calls. Branch behavior for the particular run is known, so more aggressive optimizations can be performed. But most notable is the fact that dynamic optimization can be completely compatible with the array of static optimizations already implemented in conventional compilers.

1.2 Current Approaches

There are various approaches to dynamic optimization. One approach is direct optimization of the binary executable using an executable editing mechanism. This method is typically system dependent, yet it results in a dynamic optimization infrastructure that does not have any front-end compiler requirements - any existing executable can be optimized requiring no special compilers or programming languages. Another approach involves performing optimizations on intermediate representations of the program, which are generated at the same time as the executable itself. This approach simplifies the optimization

procedure and provides for system Independence, yet it requires all software vendors to produce intermediate representations of their software. Intermediate representations, which can typically be reverse engineered to produce original source code, are not popular among software vendors. A variation of that approach, however, is the generation of an intermediate representation from the final executable, upon which optimizations are performed. This approach, while not as fast as the direct optimization of the original executable, provides for system-independent optimizations.

Dynamically optimized code can either be immediately consumed, it may be cached for use during the current or subsequent executions, or it may be permanently applied to the executable. It is evident that any optimizations that are based on the propagation of runtime constants cannot be permanently applied to the executable, as they would not be correct in subsequent runs. Yet there are several optimizations that can be permanently applied, such as rescheduling based on a hardware change.

The time at which optimizations are performed is another design decision. Most of the current research efforts focus on one of the following optimization opportunities:

- compile time, i.e. static optimization [32]
- run time, i.e. on-line dynamic optimization [6, 8, 11, 13, 25, 26]
- idle time, i.e. off-line dynamic optimization [25, 41]

All dynamic optimizations can be classified as either heavyweight or lightweight optimizations. Heavyweight optimizations make the largest amount of impact on program performance. Not surprisingly, heavyweight optimizations take the largest amount of time to perform, and therefore may not always be feasible to perform at runtime. They can, however, be performed offline, during the idle time of the computer. Lightweight optimizations, on the other hand, can typically be performed at runtime. Again, not surprisingly, most lightweight optimizations do not result in the same level of performance improvement as that of heavyweight optimizations.

The advantages and disadvantages of each of the current approaches for performing dynamic optimization will be explored in detail in Chapter 2, the background section.

1.3 Dynamic Compilation vs. Optimization

Some researchers make a sharp distinction between *dynamic optimization*, the optimization of compiled executables, and *dynamic compilation*, postponing some or all of static compilation until runtime and therefore invoking compilation on a code format that would not otherwise execute [6, 25].

Dynamic compilation differs from dynamic optimization in their means for optimization. Dynamic compilation is performed on code that was specially designed for the task. Special compilers and programming languages are required in order to generate an executable containing holes (portions of uncompiled code) that will be filled in at runtime. No special compiler or programming language is required for dynamic optimization, however, because all optimizations are performed on a working executable.

Projects involving both dynamic optimization and dynamic compilation are described in the background section, however all of the original ideas presented in this thesis focus exclusively on dynamic optimization.

1.4 Dynamic Translation vs. Optimization

Another distinction must be made between *dynamic optimization* and *dynamic translation* or *emulation* projects. Translation or emulation systems allow an executable compiled for one ISA to execute on a processor featuring another ISA. While translation systems may optimize the code that is translated, the *primary* goal is compatibility.

Several systems falling into the translation/emulation category are discussed in the related-work chapter in order to clarify the difference between these systems and the dynamic optimization system proposed later in this thesis.

1.5 Layout of the Thesis

The purpose of this thesis is to explore many old and new approaches for dynamically optimizing executables. This includes the analysis of the goals associated with dynamic optimization and the means for accomplishing the goals. Also presented is an infrastructure for the dynamic optimization of EPIC computers. In addition, several new dynamic optimization algorithms are presented which leverage features of an EPIC architecture.

Chapter 2 of this thesis contains background information. Several research projects involving dynamic optimization are discussed including the potential advantages and disadvantages of each approach. Three projects involving dynamic optimization (Section 2.1), five projects involving dynamic compilation (Section 2.2), and three projects involving dynamic translation (Section 2.3) are included in the discussion. Next, Section 2.4 discusses the current research on profiling monitors for feedback-directed optimizations. Several useful tools are available that can be leveraged in any dynamic optimization project. These tools are discussed in Section 2.5.

Chapter 3 describes the salient features of EPIC architectures and describes why EPIC is a good target for dynamic optimization.

Shortcomings discovered in many of the current research approaches led to the proposal of a new dynamic optimization system. Chapter 4 motivates and presents a high-level view of the DO dynamic optimization system. DO is a combination online/offline dynamic optimizer that is built on the solid foundation already available in the dynamic optimization research community.

One line of study that few researchers have ventured into is the development of specialized runtime optimization algorithms. Chapter 5 introduces a set of new dynamic optimization algorithms designed for implementation in EPIC architectures. After laying the groundwork for dynamic optimizations by studying the optimal time at which the optimizations should be performed, Section 5.2 goes on to introduce several new dynamic optimization algorithms that are specially designed for runtime implementation on EPIC architectures. Finally, Chapter 4 concludes that dynamic optimization is a promising new field and there is much work that can be done as future research.

Chapter 2

Background

Many research groups are currently working on Dynamic Optimization efforts in one of its many forms. Some groups work exclusively within a given instruction set architecture (ISA), gearing their optimizations toward reoptimizing for differing features of processors within a processor family or user behavior. Others work between ISAs. In this case, the final product does the job of emulating one instruction set on a different instruction-set architecture.

The purpose of the following sections is to provide an overview of some of the similar research efforts that are currently underway. A brief description of each of the approaches is mentioned, along with any notable advantages or disadvantages of the model.

2.1 Dynamic Optimization Efforts

About a dozen research groups are currently exploring dynamic optimization, compilation, or emulation techniques. Each group has a different set of approaches for accomplishing the overall goals. The approaches differ in when, how, and what is optimized. Some of the concepts contained within the approaches can be reorganized and balanced to form a new, more universal approach to dynamic optimization.

2.1.1 Dynamo - HP Labs

A group of researchers at Hewlett-Packard Laboratories developed a dynamic optimization system, which they call Dynamo. Dynamo is a caching and linking mechanism that works to optimize native executables and DLLs (dynamically-linked libraries) at run time [6].

Figure 2.1 [6] shows how Dynamo selects, optimizes and stores program fragments. As indicated in the figure, Dynamo interprets a native binary until it detects a hot trace. This is determined by observing the branches in the program. If that branch target meets certain conditions, such as being the target of a backward-taken branch, a counter associated with the target address is incremented. Only when the counter exceeds a certain threshold value is the trace deemed a hot trace (in the case of Dynamo, the threshold is 50.) The corresponding program fragment (an executable unit of trace instructions) is inserted into

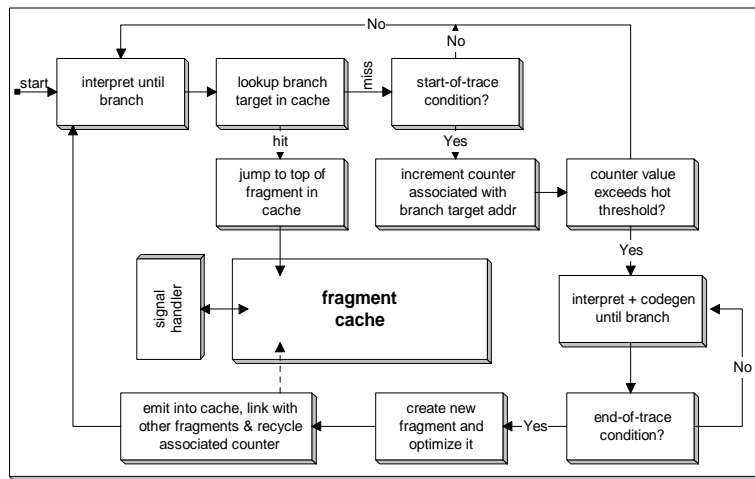


Figure 2.1: How HPL's Dynamo Works [6].

the software-based code cache, using a very speculative but simple trace-selection scheme. It is assumed that the path immediately following a hot entry point is likely to be taken in the future. Link stubs are inserted to catch any off-trace branches, in order to direct execution to the correct location, which may also be in the code cache. From that point forward, anytime the particular branch target address is encountered, Dynamo takes control and executes the optimized instructions from the fragment cache.

Dynamo ensures that time is not wasted optimizing infrequently executed fragments; only *hot* fragments are selected for optimization. This method of determining hot traces ensures that Dynamo doesn't waste time dealing with infrequently executed code. If effort were spent optimizing every program fragment, the interpretation overhead of Dynamo would be unacceptable.

HP Labs' Dynamo group points out an important distinction between their work and the solutions that are currently being developed by other research groups. First, they assume that the starting point for optimization is a native program binary. For this reason, no translation step is involved. This significantly reduces the complexity of the Dynamo solution, while allowing it to be effectively used with any existing executable. Second, the Dynamo group points out the difference in their solution and the solutions that involve *dynamic compilation*. Dynamo does not postpone any part of the compilation process until runtime. This design decision was made because the group felt that the solution should be transparent. Dynamic compilation solutions require a set of program annotations that must be inserted by the programmer, or they require the use of a new programming language that has been designed for the task of dynamic compilation. Solutions of this sort are less transparent, and are therefore excluded from the Dynamo framework. This design decision is especially notable because it allows the dynamic optimizer to bail out at any point without any adverse effects within the program itself. Bailout occurs when the amount of time spent in the fragment cache falls below a certain threshold. Dynamo suspends execution and allows the program to run directly on the underlying processor.

Another distinctive feature of Dynamo is that it is completely online, that is, all optimizations are performed at runtime. All data that is generated to aid in optimization is used within the same execution; no data is written out for later optimizations.

Yet the most notable feature of Dynamo is the performance. Even without employing any optimizations, Dynamo's methods produce speedup by storing fragments in the fragment cache. These fragments are executable traces through the code with many of the branches and procedure calls removed. Even many off-trace branches within a particular trace result in hits in the fragment cache. Taken branches normally limit fetch bandwidth, therefore the removal of procedure call and branch overheads for frequently executed paths results in an overall speedup of the code. Optimizations performed on the traces result in even higher levels of speedup.

The major advantage of Dynamo is that it is a binary-to-binary solution. No special compilation methods are needed; therefore the solution is applicable to all future applications, as well as legacy executables. Dynamo is rare in that it is a completely transparent solution. Yet the interpretation feature also requires that custom versions of some of the Dynamo code be written for each processor and operating system.

There is an entire domain of optimization, namely off-line optimization, which Dynamo does not leverage, however. The fact that Dynamo occasionally bails out of certain optimizations, but no information is stored regarding that action, leads to the conclusion that a similar attempt and failure will ensue during each of the subsequent executions. Furthermore, the optimization that was attempted will never be completed, resulting in a sub-optimal execution. If information regarding the failed attempt to optimize was maintained in an external file, the dynamic optimizer could avoid the cost of future failed attempts. This external data could also be used in order to complete optimizations off line, such as during any idle time of the operating system.

2.1.2 Morph

Zhang et al., of Harvard University developed a framework for offline dynamic optimization, known as Morph [41]. Unlike Dynamo, which performs all optimizations online (during execution), Morph collects profile information at runtime and uses that information to perform optimizations offline (after the program has exited.)

The main components of Morph are the Morph Back-end, the Morph Monitor, the Morph Editor, and the Morph Manager. The Morph Back-end is a portion of the compiler that produces the executable along with shared libraries containing all of the annotations necessary to support dynamic re-optimization. The Morph Monitor is a low-overhead profiling system that is built into the Digital UNIX operating system and serves the purpose of program monitoring. The Morph Editor is a component built into the SUIF research compiler [19] that performs optimizations on the intermediate representation produced by SUIF and outputs an executable. Finally, the Morph Manager, an off-line system component, makes decisions concerning the time at which to invoke re-optimization of programs based on the data reported by the monitor.

The Morph research group points out three necessary requirements that motivated their design. First, optimizations must occur on the particular machine where the software is actually being executed and should emphasize the usage patterns of the end user. This

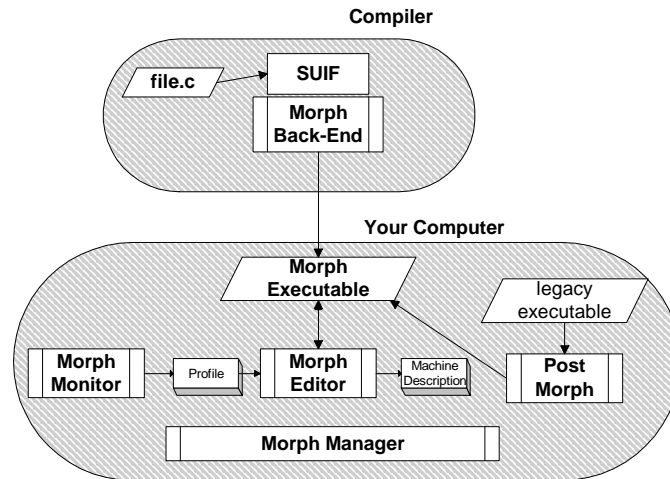


Figure 2.2: The Harvard University Morph Project [41].

requirement ensures the effectiveness of profile data. Second, no source code should be required for optimization. This requirement is clearly due to the proprietary nature of industry source code. A dynamic optimization solution that required software vendors to release their source code would not be welcomed by industry. The final requirement is that any optimization performed should be entirely transparent to an end user. A user should be able to benefit from dynamic optimization without any necessary interaction or knowledge of the optimization system. This requirement also encompasses correctness, in that optimization itself must not *break* any working code.

The three requirements listed above show that Morph is geared toward mainstream, industrial use, rather than selective research use. Morph hides complexity and insures that an end user is unaware of the reoptimization being performed, save the performance increase that will be encountered after reoptimization.

A drawback of Morph is that all optimizations are performed after the program has exited. A system such as Morph cannot leverage much of the runtime information that varies from run to run. For example, an optimization such as constant propagation, based on runtime constants, cannot be employed in the optimization process. Furthermore, the profile data collected throughout the course of an execution will be compiled into a single set of averages, and no indication of profile shifting [36] or phased behavior [37] will be evident.

Another concern is the assumption of a single user workstation environment. While Morph may be quite effective at learning the usage patterns of a single user, it could potentially hurt the performance of a multi-user program. It is not designed to be useful in an environment where a centralized binary is shared over a network, as is becoming a more and more common case.

The next drawback of Morph concerns the shared libraries containing program annotation. It is unfortunate that these annotations require compilation with the SUIF research compiler in order to be produced and cannot be generated on the fly. Morph requires

a recompilation of all existing software in order to be employed, and cannot be effectively implemented on the large quantities of legacy binaries that are available in industry. While a component of Morph known as Post Morph, which should infer Morph annotations from legacy binaries, has been mentioned briefly, details are scarce.

Finally, Morph’s program annotation is simply an intermediate representation of the entire program. Maintaining an intermediate representation of an entire program could result in severe code expansion. In addition, software vendors may not be willing to allow an intermediate representation of their software to be maintained on every users system for fear of reverse engineering.

2.1.3 Evolutionary Compilation

Sumedh Sathaye of North Carolina State University suggested many ideas for an Evolutionary Compilation system in his Ph.D. thesis [36]. His contributions include two schemes for dynamic rescheduling code for VLIW architectures: an evolutionary compiler and a dynamic rescheduler. Sathaye also contributed a technique for reducing dynamic scheduling overhead, known as the Persistent Reschedule-Page Cache (PRC).

Sathaye’s evolutionary compilation ideas were based on the notion that the usage patterns of a user evolve over time and that the code must be optimized accordingly. His dynamic rescheduling ideas focused on the different optimization opportunities available between certain processors within a single ISA.

The evolutionary compilation framework is designed to be resident in the operating system and contains a hardware-based program monitor, a loader, and an evolutionary compiler. While the application is executing, the monitoring hardware keeps track of the usage pattern of the application. If it becomes apparent that the usage pattern has changed from that for which the application was optimized, the evolutionary compiler steps in and reschedules the application to better suit the needs of the user.

The OS loader loads an application and determines if there is a generation mismatch between the object code and the machine on which it will be running. If a mismatch exists, the Dynamic Rescheduler steps in at page-fault time and performs object-code transformation with page-sized granularity. This process employs an external file, known as the object-file annotation, which was produced during the initial compile of the application.

Because the process of rescheduling an executable can be time consuming, Sathaye suggested caching the rescheduled versions of the executable in a special PRC, Persistent Rescheduled-Page Cache. This technique reduces scheduling time by storing previously translated code portions and not requiring such portions to be rescheduled and optimized every time they are encountered.

The notable feature of Sathaye’s Evolutionary Compilation system is its focus on the profile shift associated with a user’s changing usage patterns. While many of the other solutions focused exclusively on optimizing for generation mismatches, Sathaye suggests an alternate optimization stimulus.

A weakness of the system is that all retargeting is performed right at runtime, yet the rescheduled code is not maintained for subsequent executions. While a one-time slowdown for rescheduling might be acceptable, rescheduling during each execution for retargeting to the same processor is unnecessary.

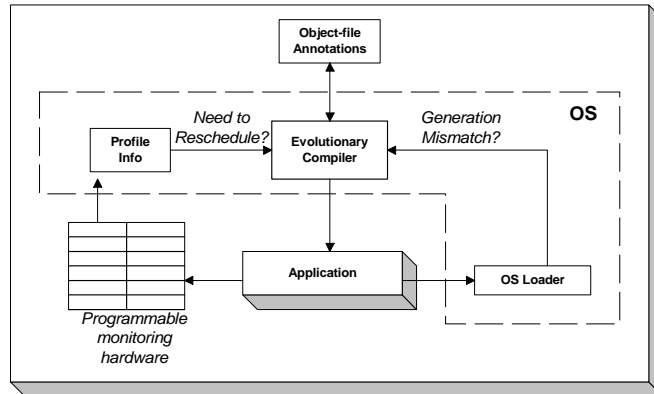


Figure 2.3: Sathaye's Evolutionary Compilation System [36].

In addition, like the Morph solution, the static compiler generates the program annotations required for rescheduling. This means that only programs compiled using a specialized compiler that generates such annotations can be rescheduled. Evolutionary compilation is not applicable to the volumes of legacy binaries in existence.

Finally, it should be noted that terminology has progressed since Sathaye's use of the phrase *evolutionary compilation*. As stated in the introduction, compilation is a term reserved for the process of generating a native executable from non-native instructions. Today, evolutionary compilation as described by Sathaye is simply referred to as *rescheduling*.

2.2 Dynamic Compilation Efforts

Dynamic compilation refers to the postponement of some or all of the static compilation process until runtime. In many cases the static compiler performs some initial compilation activities, but the creation of executable code is not performed until runtime. Dynamic compilation differs from basic dynamic optimization in this respect - the static distributed program is not executable without the dynamic compilation system in place. While dynamic optimization works with files that would otherwise execute, this is not the case with dynamic compilation. For this reason, there is a strong association between the dynamic compiler and its programs. It would not be easy to switch dynamic compilers if a program was partially compiled with a different dynamic compiler.

Several groups are currently working on projects related to dynamic compilers and dynamic compilation methodologies. Some of the research efforts are discussed in the following subsections.

2.2.1 DyC

A research group at the University of Washington has developed a rather thorough system to support dynamic compilation called DyC. DyC consists of a declarative

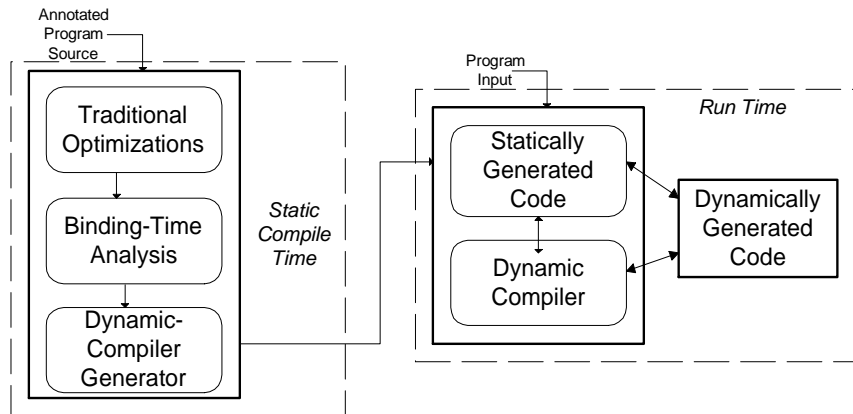


Figure 2.4: UW's DyC Dynamic Compilation System [18].

annotation language and corresponding compiler [18, 17].

The annotated language resembles C, while including the necessary instructions for providing compiler directives used in dynamic compilation. Differences in the model and a standard programming model are that DyC requires all data to be declared as static or dynamic initially, and requires dynamic compilation possibilities to be explicitly declared by the programmer.

DyC's static compiler produces an executable that contains a combination of statically compiled code and a run-time specializer for the portions of code where dynamic compilation will occur. The run-time specializer allows code to be optimized for various instances of run-time variables. Two control-flow subgraphs, called *setup code* and *template code*, replace each dynamic code region. The setup code contains static computations, while the template code contains dynamic computations. The dynamic template is created containing holes that will be filled in at dynamic compile time, once the runtime values are known. Scheduling and register allocation is performed on the two subgraphs independently, with no interference between the two.

Finally, custom dynamic compilers are implemented within the code itself, which trigger the dynamic compilation and throw away the corresponding template.

A notable advantage of the DyC model is its speed. The performance can be attributed to its lack of an intermediate format. In addition, no general analysis is performed at run time, as all decisions are made in advance and dynamic compilation is actually coded into the final executable. These two design decisions have led to notable performance gains over other research models.

A limitation of DyC that the group plans to improve upon is inadequate support for global variables and partially static data structures [17]. These limitations arise from the fact that each variable and data structure must be initially declared as static or dynamic, and no combinations are permitted.

A weakness of the DyC model, however, is that it requires the use of an entirely new language. While the system is a rather complete and adequate environment for dynamic compilation, it cannot be used with existing code. In order for mainstream use to be

possible, software developers must switch to the annotated C language.

Should the group decide to automate the annotation process and provide the functionality for DyC to work with standard programming languages, DyC may become a viable option for dynamic optimization in industry.

2.2.2 DCG

DCG, or dynamic code generation, is a system developed by Engler et al. of the Massachusetts Institute of Technology [14]. Dynamic code generation entails the idea of an executable process that creates additional executable code that is tailored to the environment on which the program is run. It is similar to the concept of *self-modifying code*, but rather than altering existing code, the existing code is augmented with additional code.

Within the DCG model, the task of code generation is postponed until run time. Instead, the intermediate representation of the `lcc` compiler is released, which will be converted to machine code just before execution. Because the intermediate representation is machine-independent, the system is completely retargetable. A single intermediate representation of a program can be used to generate code for any of a number of systems. Furthermore, the resulting executable will be optimized for the architecture on which it was created.

A single procedure is the basic unit of code generation within the DCG model. A procedure is compiled by DCG and a pointer to the code is then passed to the client. The code is invoked as an indirect call to a C procedure.

The DCG model takes care of code generation tasks including tree construction and labeling, register allocation, instruction selection, jump resolution, and binary code emission. Not included in the list of tasks is code optimization. While executable code is dynamically generated, DCG makes no efforts to support automatic global optimization or pipeline scheduling. This matter is instead "the responsibility of the client, which has access to the low-level IR specification" [14]. Despite this fact, DCG reserves eight registers for its personal use. The user can utilize these registers for a limited amount of register re-allocation, if desired.

DCG has been shown to be an efficient means for emitting code locally, however it makes no claims, nor any attempts to optimize the code it produces. The limiting factor is the fact that it produces code procedure-by-procedure, and therefore has no global awareness of optimization opportunities. This fact, combined with the fact that optimization issues are left to the client, is a clear weakness of the DCG model in the dynamic optimization domain.

2.2.3 Dynamo - Indiana University

Leone et al. of Indiana University have proposed a staged-compiler architecture for dynamic optimization which they also call Dynamo¹ [30].

Within the scheme of Dynamo, the programmer annotates a particular program with directives that specify the conditions under which dynamic optimizations should occur

¹The Dynamo project at Indiana University was actually started prior to HP Labs' Dynamo.

for a certain portion of code and the actions that should be performed in order to optimize the code.

The key feature of Indiana's Dynamo project is the staged compiler concept. During static compilation, only portions of the code that will not benefit from dynamic optimization are compiled into native code. The other portions of code are segmented into lightweight optimization candidates and heavyweight optimization candidates. Lightweight optimizations require little time to perform and only provide minor improvements to code. Heavyweight optimizations, on the other hand, are rather time consuming but typically provide substantial benefits to performance. During static compile time, heavyweight candidates are partially compiled into a high-level intermediate format. This high-level IR is suitable for more aggressive optimizations. Conversely, lightweight candidates are compiled into a mid-to-low-level IR, which is more suitable for simple optimization techniques.

A minimal amount of optimization is performed at static compile time. Selective dynamic optimization is then performed at runtime. The dynamic optimizer begins by performing candidate selection and cost-benefit analysis. Candidate selection allows the system to identify code portions that may benefit from dynamic optimization using a profiler. Cost-benefit analysis then assesses the feasibility of run-time optimization.

The most evident flaw concerns the cost-benefit analyzer. For Dynamo, this step is performed at run time. This form of analysis will most likely be extremely time-consuming and inaccurate. Not only must it accurately predict the benefit of and time required to perform a particular optimization, it must do so quickly enough not to impact the performance of the executing program. By using critical execution time, analysis of this form actually reduces the amount of dynamic optimization that can be performed at runtime.

Another drawback of IU's Dynamo is many static optimizations are forgone if it appears that they will impact the amount of runtime optimization that will be available. While many other approaches look at dynamic optimization as a complementary optimization opportunity, Dynamo sees it as a replacement for many of the static optimization opportunities.

The Dynamo approach is extremely limited in that it does not perform automatic optimization selection and instead relies on the programmer to recognize optimization opportunities and annotate the code accordingly. Furthermore, in order for existing code to benefit from the Dynamo strategy, the code must be explicitly altered. This attribute alone is such a severe limitation that it can be seen as an indication of why Dynamo may not be viewed as a universal dynamic optimization solution.

2.2.4 Runtime Optimization

In an effort labeled Dynamic Runtime Optimization, Kistler, et al. of the University of California at Irvine suggests moving all program optimization to runtime [25]. At compile time, he suggests that the role of the compiler should be to produce an object file that contains a generic tree-based intermediate representation of the program, rather than native machine instructions. No optimizations are performed on the intermediate format, as that would jeopardize its portability. This would allow the intermediate format to be issued to any of a number of processors, which would then optimize the object file to take advantage of the particular features of the processor itself. This optimization would all be done at

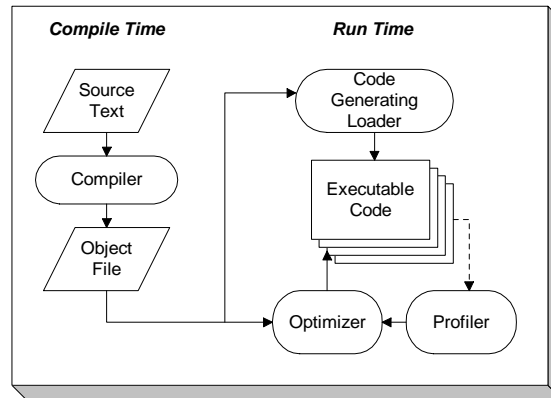


Figure 2.5: Kistler's Dynamic Runtime Optimization System [25].

runtime, however, utilizing the idle time of the program.

Kistler suggests a new system architecture that consists of four key elements: a compiler, a dynamic code-generating loader, an adaptive profiler, and a dynamic optimizer.

The role of the compiler is to generate an intermediate representation object file from the original source code. It is not the role of the compiler, however, to perform any static optimizations, as that may obstruct the opportunity for dynamic runtime optimizations.

The dynamic code-generating loader then transforms the intermediate representation into native machine code at load time, then executes the given code. During the execution period, the adaptive profiler observes the runtime behavior and gathers all of the necessary information required for future dynamic optimization decisions. Finally, the dynamic optimizer periodically optimizes code portions that are executed most frequently. The intermediate code representation is optimized in the background during the idle time of the operating system.

One of the major advantages of this runtime optimization model is that it allows the dynamic optimizer to be aware of the time-consuming portions of the code (via the profiler) and to focus optimization efforts on such portions. This is clearly superior to the static optimization model, which applies optimizations uniformly to all code portions, regardless of usage patterns.

The major disadvantage of this system architecture is that a great deal of strain is put on the runtime system. In typical systems, the only task performed at runtime is the execution of native code on the processor. In this system, however, the runtime system takes on the added responsibility of code generation, profiling, and dynamic optimization. Moving this much functionality to the runtime system may not result in the desired performance gains.

In addition, this model increases code size significantly due to the intermediate representation of the code that must be kept around for use in the dynamic optimizer.

This form of runtime optimization is not applicable to the existing code base. It would require software vendors to either release their source code (not an option), or compile

their software into the proprietary intermediate format. Furthermore, software vendors may be leery of the intermediate representation of their code that will be kept around. If there is any possibility of decompilation of the IR, vendors will not be willing to implement the optimizer for fear of code piracy. Unfortunately, this model requires 100% compliance by software vendors in order to be effective.

2.2.5 Tempo

In 1996, Charles Consel of the University of Rennes / Irisa, France introduced an infrastructure, which he called Tempo, upon which much of the DyC project of the University of Washington was based [9].

Consel was the first to automate the process of run-time specialization, where a template is produced containing holes that correspond to each of the values that are constant at run time. Using formal methods, Consel proved the correctness of his proposed techniques for a variety of languages and developed Tempo, a prototype version of his run-time specializer based in the C programming language.

At compile time, a tree grammar is produced, which is an approximation of the possible program specializations. This tree grammar is produced using programmer annotations in the code that specify the program invariants. The tree grammar is then converted into a set of templates. At run time, a run-time specializer selects the appropriate templates, computes invariants and fills them into the template holes, and relocates jump targets.

This method for dealing with constants known at runtime was revolutionary for its time. When compared to other pre-1995 run-time specializers, Tempo possessed the following notable traits. It was the first automated specializer of its time. While other research groups had employed methods similar to templatizing code, all such templates were created by hand and therefore suffered from the risk of programmer error. Because the Tempo compiler produced the templates automatically, the risk of error was greatly reduced. Second, Tempo's correctness was formally proven and published [9]. Third, although a prototype was developed in C, the details of Tempo were applicable to any of a number of programming languages. Fourth, this model was the first of its kind to be portable. And finally, Tempo was shown to be an efficient means of run-time specialization. The code produced required as little as three runs to amortize the cost of specialization, resulting in code that ran five times faster.

On the negative side, Tempo addressed only one of the numerous goals of dynamic optimization. Tempo makes no attempt to deal with issues such as rescheduling code for differing architectures, or register reallocation.

Furthermore, the one goal that Tempo did address, it did so in a manner that required a great deal of programmer effort. While the task of template creation was in fact automated, the task of determining potential templatized portions was still the duty of the programmer. This not only required effort and knowledge on behalf of the programmer, but it introduced the potential for error and/or suboptimal specialization.

Name	Dynamic Optimization	Dynamic Compilation	Optimization Time	User Effort
DyC		X	Runtime	High
Morph	X		Offline	Med
Dynamo (HPL)	X		Runtime	Low
Dynamo (Indiana)		X		High
DCG		X	No Opt	High
Tempo		X		High

Table 2.1: Summary of past approaches.

2.2.6 Summary

The dynamic compilation and optimization methods presented in this section can be summarized in Table 2.1. Several of the research groups employed dynamic compilation, which is an interesting approach but it requires a great deal of software development effort. New compilers and programming languages are required in order to employ such an approach. Other groups chose the dynamic optimization route, however approaches varied widely. The approaches ranged from a high amount of user effort required, to completely transparent approaches that required no user effort at all. Most approaches perform optimizations at runtime and very few perform any optimizations offline.

2.3 Translation and Emulation Efforts

Several research groups have undertaken the task of emulating a program compiled for one ISA in an environment that uses a different ISA. While IBM developed an emulator for executing x86 instructions in a VLIW environment, Transmeta developed a general approach to program translation. Future efforts that provide for compatibility with IA-64 will soon be underway.

2.3.1 DAISY

A group at IBM's T.J. Watson Research Center has developed a system that they call DAISY: Dynamically Architected Instruction Set from Yorktown. DAISY focuses on providing complete compatibility between architectures [13].

While most DAISY documentation focuses on VLIW as the target architecture, with modifications DAISY could be used to translate code to execute on processors using the new EPIC instruction set architecture.

As each code fragment is executed for the first time, DAISY translates the fragment into a simple RISC-like intermediate representation. This translation occurs at instruction cache miss time. The translated version of the code is then stored in a caching emulator, which is simply a portion of main memory that is not visible to the old architecture and resembles a cache. Because the translation is cached, the process of translating each fragment must only occur once.

Within DAISY, precise exceptions are handled by scheduling VLIW instructions in the same order as they occurred in the original binary. Unfortunately, this scheduling mechanism will result in code that is far from optimized.

DAISY provides 100% architectural compatibility between ILP architectures and other existing architectures. It also provides fast emulation using clever caching techniques. Such caching techniques would not be necessary, however if the new translated version of the code fragments would actually replace the old code.

2.3.2 Crusoe

In January 2000, Transmeta announced the Crusoe package [26]. Comprised as a combination of a low-power VLIW chip and specialized Code MorphingTM software, Crusoe entered the mobile computing market offering a microprocessor that achieved notable performance while using a fraction of the power.

While the core Crusoe processor is a 4-wide VLIW, it is able to run all legacy x86 code using the Code MorphingTM software, which translates x86 instructions into the native instruction set on the fly. The savings in power came from the fact that not all attributes of the x86 architecture needed to be supported in hardware. An entirely new, simple processor could be used.

The Crusoe technology essentially eliminates the ties between the x86 ISA and the underlying hardware. Furthermore, as upgrades to the Crusoe processor are released, the user need only replace the Code MorphingTM software, which is specialized to the new processor. Even dramatic changes in the processor will not require any of the software (aside from the code-morphing software) to be recompiled.

The more advanced of the two Crusoe processors introduced in January 2000, can also perform LongRunTM power management. Unlike many mobile processors which have two clock frequency settings for power savings, full speed or off, the Crusoe processor can dynamically analyze the program(s) being run and adjust the clock frequency to any frequency between maximum and off. Crusoe can also vary the voltage used by the processor as the clock frequency is varied. These adjustments of clock speed and voltage result in cubic reductions in power since power varies linearly with clock speed and quadratically with voltage.

The most ingenious feature of the Crusoe system, is that the Code MorphingTM software could be upgraded to not only translate x86 code, but Sparc, Alpha, HP-UX, and IA-64 code as well. This would result in a single system that could transparently execute any legacy code compiled for any architecture without the user knowing or caring about compatibility issues.

The drawback of the Crusoe system is that the translation step is not free; some amount of time must be spent executing the code-morphing software. The translated groups of instructions are maintained in a translation cache, however, amortizing the cost of the translation step. Because the translated code stored in the translation cache is optimized, it can potentially execute faster than the original code.

While the power saving features of the Crusoe processor makes it an competitive option for mobile computing, the translation and optimization features could be employed in the general-purpose (desktop) computing market. The potential for compatibility among all

instruction-set architectures is a goal that the entire computing community could certainly benefit from.

2.3.3 FX!32

Compaq (Digital) Corporation introduced a combined emulator and translator that allows x86 applications to run on an Alpha using the Windows NT 4.0 operating system [22]. This system, called FX!32, combines run-time emulation with an off-line optimization system in order to truly distinguish itself from other emulation systems, such as DAISY or Crusoe.

While all of the emulation systems described in this section perform the task of executing non-native instructions on a particular processor, FX!32 employs advanced background dynamic optimization to optimize the Alpha code that is produced during the emulation stage using profiling information gathered at runtime.

The design goals of the background optimization system were user transparency and high performance. To ensure transparency, FX!32 had to be certain that no user interaction was necessary during optimization and that the native code produced was complete and correct. These restrictions resulted in an optimization algorithm that relied on no assumptions or manual interaction of any kind. To ensure the best possible performance, FX!32 leverages many modern advances in compiler optimizations.

A major optimization leveraged by FX!32 is the concept of global optimization. While many other dynamic optimizations are limited to discrete elements such as basic blocks, FX!32 breaks this limitation and is able to perform global optimizations that were currently implemented only during static compile time. This method produces an Alpha executable that performs at least 70% as well as the ideal performance value, which is determined by directly compiling the original source code into an Alpha executable.

2.4 Profiling Monitors For Feedback-Directed Optimization

An important feature of any dynamic optimization system is its ability leverage profile information about a user to guide runtime optimizations. This allows the software to perform well for the way it is typically used. This concept is known as feedback-directed optimization. The feedback from the usage patterns guides the optimization process.

The question of *how* to monitor usage statistics for a particular piece of software without hurting software performance is an open research area. The monitoring could be performed in the operating system, such as a daemon that frequently polls the software and gathers statistics much like the monitoring scheme suggested in Thomas Kistler's dynamic run-time optimization system in Section 2.2.4. The problem with a software-monitoring scheme such as this is that it carries a lot of overhead. The scheme requires a system call to be performed at given intervals in order for statistics to be gathered. The overhead associated with software monitoring may outweigh the benefits gained from dynamic optimization.

Another alternative is to provide dedicated profiling hardware. Since all profiling tasks are performed by the dedicated hardware, there is no performance impact on the

running program. The drawback of hardware-based profiling is the expense incurred in adding specialized hardware. Convincing processor manufacturers to dedicate silicon to a task such as profiling may not initially be a welcomed concept. This silicon could be used for a more universal application, such as speeding up the clock or adding a function unit. Furthermore, upgrading the monitoring hardware would require a new processor. Hardware monitoring is much less flexible or upgradeable than its software counterpart.

Several research groups have looked into the problem of program monitoring and profiling. While most groups agree that hardware profiling is the most feasible form of program profiling, their approach and scope differs greatly.

2.4.1 ProfileMe

In order to overcome many of the limitations of profiling in the out-of-order processor domain, Digital Equipment Corporation introduced a novel profiling mechanism, called ProfileMe [12]. Rather than employing event counters that monitor specific events such as the number of branch predictions and causing an exception when the counters overflow, the group chose to randomly choose instructions and gather all pertinent information regarding the specific instruction.

The move to instruction sampling arose from problems noted in previous monitors. In order to optimize a program, it is necessary to discern the particular instruction(s) causing an unfavorable event such as a branch misprediction. In previous models, accurate event counts can be maintained, but the method for indicating the instruction causing each event is rarely correct. Out-of-order instruction execution complicated this process, resulting in a nearly random distribution of causing instructions. Speculative execution further complicates the process, as speculative instructions were ignored in previous models. ProfileMe permits the analysis of which instructions are aborting and why, while other sampling methods ignore aborted instructions.

ProfileMe is a hardware/software approach to accurately track events and their causes. This is done in two ways. The first method is the *instruction sampling technique*. This technique selects a random instruction and tracks all information about the instruction, such as cache miss rates. Another sampling technique can also be employed. This technique, known as the *paired sampling technique*, collects data about the interaction of other instructions with the random selected instruction.

The major advantages of ProfileMe over techniques such as hardware event counters are:

- (1) It collects complete information about each instruction, rather than sampling a small number of events at a time;
- (2) it accurately attributes events to instructions;
- (3) it collects information about all instructions, including instructions in uninterruptible sections of code; and
- (4) it collects information about useful concurrency, thus helping to pinpoint real bottlenecks [12].

The ProfileMe group chose sampling as a means of profiling rather than some of the other methods, such as simulation or instrumentation, for three reasons. First, sampling works on unmodified programs. No additional instructions need to be inserted

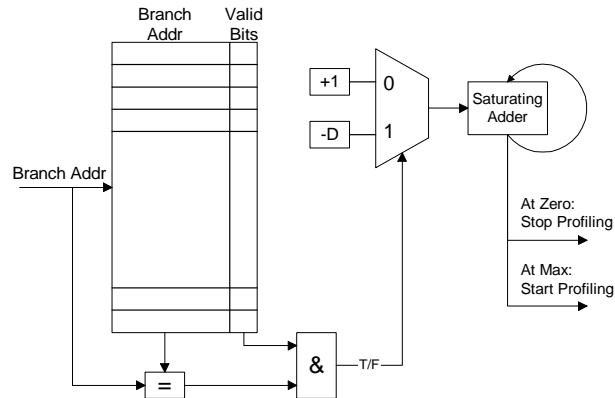


Figure 2.6: Illinois' Monitor Table Hardware for Hot-Spot Detection.

into the program in order for sampling to proceed. Second, sampling can be used to profile entire systems. Finally, sampling requires very low overhead. The overhead can be directly correlated to the sampling rate. Reducing the sampling interval can therefore directly reduce the overhead caused by sampling.

2.4.2 Profiling for Hot Spots

Wen-mei Hwu's group at the University of Illinois extends the typical profiling scheme by also recording the relative importance of the basic blocks encountered [34]. They define a hot spot of code to be a small section of code that is executed very frequently.

The importance of hot spot determination lies in the realm of dynamic optimization. A frequently executed portion of code that contains a small static footprint, as in the case of hot spots, is a prime candidate for runtime optimization. Due to time restrictions, not all optimizations can be performed at runtime. Therefore potential optimizations must be prioritized. In the case of hot spots, a small amount of time may be required for optimization, compared to the long period of time that will subsequently be spent executing the code. Hot spots should therefore be the top priority for runtime optimizations.

The Illinois group proposes three components to aid in the tasks of hot spot detection and its corresponding dynamic optimization. These components include a *Hot Spot Detector*, a *Monitor Table*, and *Operating System Support*. The hot spot detector can be implemented in hardware to remove overhead. Its main task is to locate the blocks of code that are most frequently executed. The monitor table is a collection of previously identified and optimized hot spots. It is the task of the monitor table to handle the replacement scheme of optimized hot spot code. If it is determined that a particular hot spot is no longer frequently executed, it should be replaced by a better runtime-optimization candidate. Finally, operating system support is necessary for adding hot spots to the monitor table and gathering blocks into a region that can be passed to the runtime optimizer.

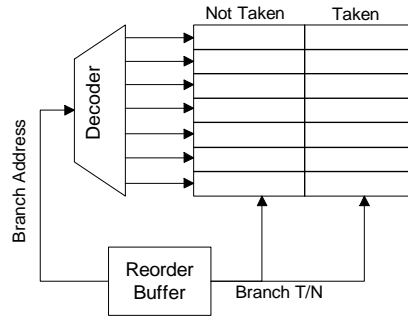


Figure 2.7: The Profile Buffer Hardware [10].

2.4.3 The Profile Buffer

Because of the large overhead that can be encountered during profiling, a group at North Carolina State University proposed a profile buffer for maintaining accurate profile information with low overhead [10].

The profile buffer hardware consists of a decoder and a set of branch counters. Each row in the buffer has two buffer entries associated with it, one is a counter that maintains the number of times the branch was taken, the other maintains the number of times the branch was not taken.

As a branch is encountered, it is converted to an index into the buffer table. Once the outcome of the branch is known, the corresponding entry in the buffer table is updated. After a certain amount of time and before the profile buffer fills, the contents are spilled to memory. In order to hide the latency of the buffer spill, the action is performed during a context switch.

Due to the fact that the profile buffer restricts profiling to include only branch outcomes, it alone is not useful in the context of dynamic optimization. Dynamic optimization requires a great deal more profile information in order to perform any worthwhile optimizations. If the profile buffer was included in a large set of monitoring hardware, however, its low-overhead scheme could be very useful to runtime optimization where performance is the top priority.

2.4.4 Itanium Monitors

Chapters 4 and 5 of this thesis focus on dynamic optimization in the realm of EPIC architectures. For this reason, the particular monitoring features that will be available in the first chips employing the IA-64 architecture are of particular interest.

Intel Corporation recently disclosed the specific features that will be incorporated into the Itanium processor line that will radically simplify the task of monitoring and profiling executing programs [23]. The new features focus on locating performance bottlenecks.

Specifically, the performance monitors that will be built in provide four levels of profiling functionality. First, the Itanium will have the ability to monitor a large number of event rates, such as cycles per instruction, miss rates for the caches and translation

lookaside buffer, branch misprediction rates, or any other rate that need be monitored. Second, Itanium provides support for breaking the total execution time of a program down into one of four components: branch related, execution related, data access, and instruction access. Third, hardware support will be available for monitoring specific pieces of code. This will allow the programmer to select certain address or data ranges, then zoom in on the specific activities occurring in that range. Finally, Itanium will be equipped with certain *event address registers* that can be used for maintaining sampling counts, such as the number of branch mispredictions.

The specific monitoring features that will be resident in the Itanium processor will clearly be of extreme importance to any developer of a dynamic optimization system for IA-64.

More importantly, these features indicate the processor designers' support and approval of future dynamic optimization efforts. Eventually, the interest in hardware-based performance monitors by the optimization community will lead to increased hardware support within the processor development community. The expressed interest is therefore vital for the continuous development of advanced hardware monitoring features.

2.4.5 Summary

Profiling monitors provide invaluable information to any dynamic optimization system. Fortunately, several research groups have focused on providing fast, effective profiling in both hardware and software. Furthermore, industry is beginning to show support for hardware monitoring by incorporating many profiling features right into their microprocessors. Eventually, all hardware features are attributed to the demand for such features. As we express more and more interest in monitoring features, chip designers are more apt to incorporate those features into future designs.

2.5 Useful Tools for Dynamic Optimization

A number of tools are available to make the complex tasks involved in optimizing executable files much simpler. In addition, because these tools have been proven correct, they aid in reducing the risk of error in the end product.

2.5.1 DynInstAPI

A research group at the University of Maryland - College Park has developed an API, or application program interface, that can be used for runtime code generation called Dyninst API [7].

Figure 2.8 [2] shows an example of the runtime code generator API being used in several lines of C code. The DynInstAPI could be very useful to a dynamic optimizer because it could be used to insert code into an executable that would trigger the dynamic optimization process after a given event occurs (such as the detection of a profile shift.)

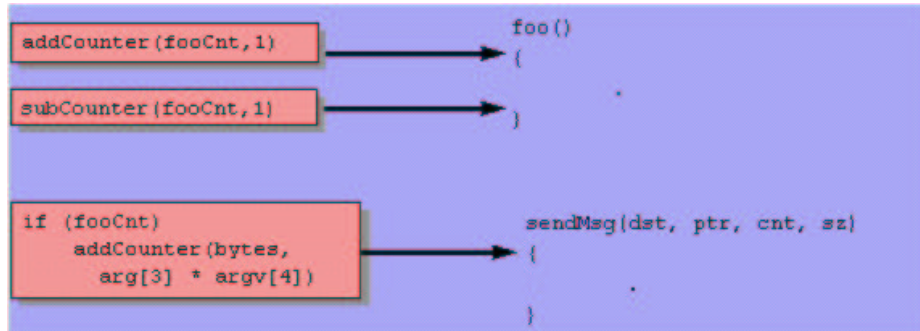


Figure 2.8: Runtime Code Insertion via DynInstAPI.

2.5.2 EEL

An executable editing library, EEL, has been developed at the University of Wisconsin that hides much of the complexity of editing anything from object files to fully-linked executables. The library also hides any system-dependent details of editing executables. EEL is currently available for the Solaris operating system [28].

Using EEL, an executable may be edited at the level of a routine, a control-flow graph, an instruction, a snippet, or an entire executable, which consists of routines and data. EEL provides comprehensive control-flow and data-flow analysis for CFGs. It automatically handles required modifications to calls, branches, and jumps necessary for CFG adjustment.

EEL could be extremely useful to a dynamic optimization system because it has already been proven effective for global register allocation and interprocedural analysis. While it is not designed to handle run-time editing, it could be useful for heavyweight offline optimization.

2.5.3 OM

The OM project at Digital Research focuses on binary editing and its applications. OM is a binary editor designed for code optimization, instrumentation and translation. It reads an executable, converts it to an intermediate representation, edits the IR, then converts the IR back into an executable [38].

OM's major difference from EEL is that it allows conversion to a different binary format from that which originally existed. Because all executables are converted into a generic intermediate representation, any executable format can be generated.

The use of the IR makes this tool much slower than EEL, and would probably only be useful for converting between different ISAs. Unfortunately, this tool is not available to those outside of Digital.

Chapter 3

Salient Features of EPIC Architectures

The release of Intel's new EPIC instruction-set architecture opens the doors for a new set of run-time and post-run-time optimizations. Such optimizations can focus on the idiosyncrasies of the new architecture in order to produce optimizations for performance and scalability that is not possible at compile time and are exclusive to the EPIC architecture.

A discerning factor of the EPIC architecture in comparison to its superscalar predecessor is that instruction scheduling is no longer done on the fly at runtime, but now becomes a compile-time task. This means that instructions are scheduled without any knowledge of user inputs. The compiler must now predict the outcome of branches before learning of any runtime constants. In cases such as these, it would be quite advantageous to employ a runtime optimizer that uses runtime information to reoptimize the code scheduled by the static compiler.

3.1 New Features of IA-64

Several new features embody the new 64-bit architecture that will be used with the new Itanium chips starting in the second half of 2000 [23]. A great many differences can be noted between the new IA-64 architecture and its predecessor, which has been an industry usage leader for personal computers for many years - the x86 architecture.

The overall goal of the IA-64 architecture was to produce an instruction-set architecture that allowed for the necessary hardware expansion that would occur over the new few decades, while also providing industry-leading performance.

In designing the IA-64 architecture, six goals were emphasized [23]. The designers made a special attempt to (1) increase instruction-level parallelism, (2) better manage memory latencies, (3) improve branch handling and management of branch resources, (4) deliver exceptional floating point performance, (5) reduce procedure call overhead, and (6) provide massive resources. Such goals were realized by incorporating several notable features into the ISA. Several distinguishing features of IA-64 were described in [23] and are summarized in the following sections.

3.1.1 Large Register File

First, the IA-64 architecture provides a much larger register file than most of the conventional ISAs. There are 128 64-bit general-purpose registers, 128 82-bit floating-point registers, 64 one-bit predicate registers, and eight 64-bit branch registers. This massive register file allows multiple computations to be performed in parallel with only compulsory memory accesses.

3.1.2 IA-64 Instructions

IA-64 instructions are grouped into three 41-bit instructions and a 5-bit template, which make up a *bundle* as shown in Figure 3.1. The template bits represent the type of instructions that make up the bundle. Instruction types fall into one of six predefined groups shown in Table 3.1. Each of the instruction groups is sent to one of the four types of execution units: instruction, memory, floating point, and branch. The set of allowable templates is shown in Table 3.2.

IA-64 introduces the concept of *instruction groups*. An instruction group is made up of a variable number of instruction bundles, followed by a stop bit. The stop bit is a signal to the processor that the instructions following the stop bit cannot be executed in parallel with the instructions preceding the stop bit due to a dependency. The stop bit is the fifth bit of the instruction template as shown in Figure 3.1.

The first implementation of IA-64, codenamed Merced, is the Itanium processor. During any given cycle, two instruction bundles are issued to the function units for execution in Merced. This results in up to six instructions issued to the function units during a single cycle.

3.1.3 Explicit Parallelism

Because stop bits must be inserted between dependent instructions, it is the duty of the compiler to determine and explicitly express the parallelism available in a particular program. After determining each of the read after write (RAW) and write after write (WAW) hazards, the compiler schedules the instructions into a set of instruction groups which can be issued to the processor in order. No instruction reordering is performed at runtime; the schedule that is determined at compile time is maintained throughout execution. This compile time scheduling allows instructions to be issued without hardware dependence checks, which is a key factor in the performance of IA-64.

3.1.4 Memory Access and Speculation

Like other load-store architectures, IA-64 allows memory access only through explicit load and store instructions¹. Through an effective use of speculation, all memory latency can furthermore be reduced. Speculation allows an instruction to execute before the processor knows if it is necessary. A load can be issued before all prior store addresses

¹An exception to this rule applies in the case of a special semaphore instruction, which allows limited memory access.

```

br    L1                //cycle 0
ld    r3 = [r5];;      //cycle 1
shr   r7 = r3, r87     //cycle 3

```

Figure 3.2: An IA-64 Instruction Sequence without Control Speculative Loads.

```

ld.s  r3    = [r5]      //advanced load
      ...
br    L1;;            //cycle 0
chk.s r3    //cycle 1
shr   r7    = r3, r87  //cycle 1

```

Figure 3.3: An IA-64 Instruction Sequence with Control Speculative Loads.

are known. This hides the latency of the load instruction, allows the data from the load to be available as soon as it is needed.

Two types of speculation exist: control speculation and data speculation.

3.1.5 Control Speculation

Control speculation [32], was also incorporated into IA-64. Control speculation occurs in IA-64 as loads and their respective uses are moved above branch boundaries. A special NaT² bit exists for each register in the register file. This bit allows control speculation to proceed without causing unnecessary page faults or other exceptions yet also maintaining precise exceptions. If a speculative load causes an exception, the exception is not handled until it is known that the load was actually necessary. Instead of handling the exception, the NaT bit is set on the destination register of the load. This NaT bit is propagated to all of the uses of the load. It is not until an operation is executed non-speculatively that the NaT bit is checked and the corresponding exception is handled.

Figures 3.2 and 3.3 are examples of the cycle time difference between an instruction sequence with vs. without control speculative loads.

3.1.6 Data Speculation

Data speculation occurs when loads are moved above memory references that may potentially overlap with the memory address of the load. For example, in the sequence shown in Figure 3.4, current compilers could not move the load above its preceding store, because it is not known whether register 55 and register 5 contain the same memory addresses.

The load instruction in Figure 3.4 is replaced by the advanced load and load check instructions in Figure 3.5. In this case, data speculative loads removed two cycles from the execution time of the code fragment.

²NaT stands for Not a Thing, and is used to signify an invalid speculative register.

```

st    [r55] = r45      //cycle 0
ld    r3    = [r5];;   //cycle 0
shr   r7    = r3, r87  //cycle 2

```

Figure 3.4: An IA-64 Instruction Sequence without Data Speculative Loads.

```

ld.a  r3    = [r5]     //advanced load
      ...
st    [r55] = r45      //cycle 0
ld.c  r3    = [r5]     //cycle 0 - check
shr   r7    = r3, r87  //cycle 0

```

Figure 3.5: An IA-64 Instruction Sequence with Data Speculative Loads.

3.1.7 Predication

In order to overcome the major performance penalty incurred by branches, IA-64 uses predication as a method for removing a substantial number of the branches within code. The concept behind predicated instructions is simple. Instructions may have a predicate associated with them, which is a condition that may be evaluated as true or false. If the predicate is evaluated as true, the instruction proceeds normally; if the predicate is false, the results of the execution are not committed. Predicate registers are one-bit boolean registers that can be associated with any given instruction. The example in Figures 3.6 and 3.7 shows the conversion of a set of C instructions with branches into the corresponding predicated IA-64 code.

The advantage of the predicated instruction sequence is that since the branches are removed, it is no longer necessary to produce changes to the program counter that result in pipeline stalls. The code can essentially be executed in sequence as if no branches were present, resulting in a major performance boost. Yet, the drawback of predication is that a balance between control flow and predication should be attempted. Over-predication of instructions will result in a high percentage of code being executed whose results will be thrown away due to false predicates.

3.1.8 Stacked Registers (Register Windows)

The IA-64 register file is organized in a manner that results in a large reduction in the overhead experienced in a context switch such as a procedure call.

As a procedure is called, the first 32 general-purpose registers (GR0-GR31) are automatically visible and are referred to as static or global registers. The procedure may then allocate up to 96 additional registers (GR32-GR127), known as the stacked registers to be used as input, output, or global variables. The set of input/output/local registers for a particular procedure is referred to as the procedure's *stack frame*. As additional procedures are called, the output registers of the calling procedure are overlapped with the

```

if (a) {
    b = c + d;
}
if (e) {
    h = i + j;
}

```

Figure 3.6: Sample C Code with Branches.

```

cmp.ne  p1, p2 = a, r0    //p1 ← a ≠ 0
cmp.ne  p3, p4 = e, 0;;   //p3 ← e ≠ 0
(p1) add  b = c, d        //if a ≠ 0 then add
(p3) sub  h = i, j        //if e ≠ 0 then sub

```

Figure 3.7: Corresponding IA-64 Predicated Instructions.

input registers of the called procedure. The stacked register concept allows parameters to be passed between procedures with little to no overhead.

3.1.9 Software Pipelining Support

Software pipelining is a compilation method that is frequently used to improve the parallelism and performance of loops. During software pipelining, loops may be unrolled and may therefore be resident in the execution pipeline concurrently with other iterations of the same loop.

The IA-64 architecture provides much-needed support for software pipelining, which reduces the likelihood of code expansion and cache misses that plagued earlier VLIW efforts. The support includes dedicated instructions and a rotating register scheme.

Rotating registers refers to the design principle where registers are rotated by one position each time an iteration of a loop is initiated. Rotating registers allows for the removal of anti- and output dependencies across loop iterations. Without this feature, it would be necessary to unroll loops and apply register renaming. The general-purpose, floating-point, and predicate registers all support register rotation. This scheme simplifies the software-pipelining algorithm allowing a reduction in necessary instructions.

3.1.10 Register Stack Engine

Unlike previous architectures, IA-64 maintains dedicated hardware, known as the register stack engine, which performs the task of moving data from memory to the registers and vice versa. No program intervention or processor resources are wasted spilling data from the registers, resulting in notable performance improvements.

3.2 Summary

Numerous state-of-the-art features are present in the IA-64 instruction-set architecture. Improvements have been made to the resources and algorithms available to handle many of the bottlenecks of previous architectures. Some of the bottlenecks included a limited-sized register file, and low potential for achieving high levels of instruction-level parallelism.

A great deal of effort has also been placed on producing an ISA that will withstand the test of time by foreseeing the many upgrades that may be performed in the years to come. One of the major weaknesses of its predecessor - the x86 architecture, was that it had reached its limits in terms of expandability. The final IA-64 design includes a great deal of improvements that have resulted in the potential for a massive performance increase and scalability.

Chapter 4

DO: A Dynamic Optimization Infrastructure

A major limitation of many of the proposed solutions regarding dynamic optimization is that the methods are not applicable to existing precompiled software. Programmers must either flag certain portions of the code as dynamically optimizable, or software vendors must compile their code into an intermediate format or using a proprietary compiler in order for dynamic optimization to be possible. This results in optimizations that are only as effective as the programmer his/herself and a solution that is only as effective as vendors are willing to provide. The solutions require an unrealistic full compliance on the part of the programmers and software vendors in order to be effective. While it would certainly be advantageous if we could guarantee such compliance, it is often not the case.

Several project goals should be established before designing a dynamic optimization infrastructure. In our case we have chosen compatibility, transparency, and performance as our most important goals. Each goal is described in the following sections.

The Compatibility Goal. A goal overlooked by many of the research groups is finding a solution that is just as applicable to past development (i.e. legacy code) as it is to future software development [14, 18, 30]. This does not preclude the use of new compiler-inserted hints; it merely restricts them from being a requirement. Optimization hints inserted by the compiler should be helpful, but *not* necessary, as this would adversely impact the effectiveness of this model on legacy code. Compiler hints could include information regarding register allocation that would aid in future reallocation.

In DO, program annotation is either produced by the compiler back-end or it is generated as an offline process. Yet many basic lightweight optimizations can be performed on code containing no annotation. This fact can often be leveraged during program optimization while no annotation information is available (typically during the first execution of a legacy binary).

The Transparency Goal. A user need not have any awareness of a dynamic optimization system in order to benefit from optimized software. Dynamic optimization should be automatically invoked on any piece of user-level software that is started on a machine. Furthermore, the entire optimization process should not require any interaction from the user while it is running. These are all realistic expectations of any effective dynamic opti-

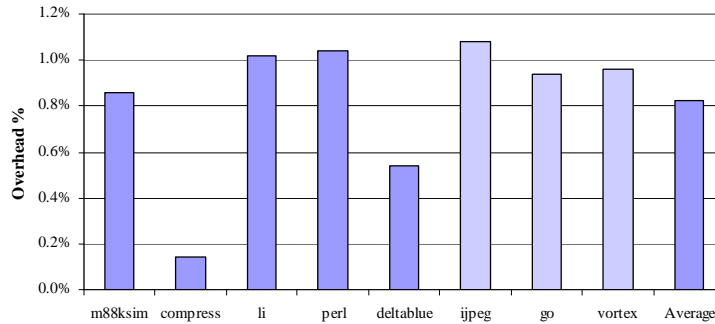


Figure 4.1: Dynamo Overhead as a Percentage of Execution Time.

mization system. To design a system that is anything but transparent from the viewpoint of an end user would result in a system that is only as effective as its user. For this reason, transparency is a top priority in the DO infrastructure.

The Performance Goal. The lower bound for performance in a dynamic optimization system should be the performance achieved without dynamic optimization. However, there are many factors that come into play with such a system. While an initial slowdown may occur, this may be acceptable if that slowdown is amortized by the speedup during successive executions. Furthermore, the actual optimizations performed play an important role in the performance improvements achieved, so it would be expected that the performance of a particular dynamic optimization infrastructure will improve as more research is done in the area of dynamic optimization algorithms.

4.1 An Extension of Dynamo

In order to meet the goals of compatibility, transparency and performance, a new dynamic optimization infrastructure is proposed. This infrastructure, called DO, is completely software resident, but extends beyond the boundaries of existing dynamic optimizers by providing for offline optimization and retaining information regarding previous dynamic optimization runs in the form of DLLs (dynamically-linked libraries). DO exists as user-level software on a system employing the explicitly parallel instruction computer (EPIC) ISA. This opens the doors for a variety of optimizations available only in the EPIC ISA, such as employing the stacked register allocation scheme for dynamic register reallocation.

Because a large number of dynamic optimization systems have been developed over the years, it is not necessary or even feasible to attempt to build an entirely new dynamic optimization system from scratch in order to make significant contributions to the world of dynamic optimization. Instead, a better approach is to leverage some of the existing work in dynamic optimization infrastructure development and make various extensions that are seen fit due to existing weaknesses in the design or to compensate for advances in dynamic optimization infrastructure research.

With this in mind, the DO infrastructure builds on the existing HP Labs Dynamo codebase, yet notable extensions are incorporated. First of all, while Dynamo worked

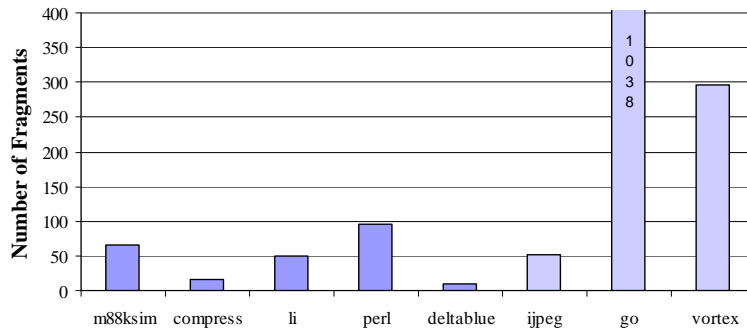


Figure 4.2: The Smallest Number of Fragments that Account for 90% of the Total Execution Time.

exclusively on a PA-RISC architecture employing the HP-UX operating system, DO will be based on a version of Dynamo ported to the IA-64 architecture with versions available for both the Linux and Windows operating systems.

4.2 Performance Improvements Achieved by HPL’s Dynamo

Two factors motivated using Dynamo as a starting point for the DO infrastructure. First, Dynamo shared many of the underlying goals described in Section 4.1. The goals of transparency, compatibility and performance were already incorporated into Dynamo. The most promising feature of Dynamo, however, was the performance improvements achieved. Dynamo produced an average of 11% speedup on the SpecInt95 benchmarks with less than 1% overhead.

Figure 4.1 shows the overhead of running Dynamo on the SpecInt95 benchmarks [6]. The average overhead incurred by Dynamo was less than 1% of the overall execution time. The lightly shaded bars represent benchmarks that Dynamo deemed *ill behaved* and bailed out of the optimization attempts. Ill-behaved programs are programs that do not spend significant amounts of time in a stable working set. By the time Dynamo has identified program hot spots, the working set changes.

As stated before, Dynamo works at the granularity level of program fragments, which is an executable unit of trace instructions. Figure 4.2 shows the smallest number of program fragments that are necessary to account for 90% of the total execution time of the SpecInt95 benchmarks [6].

Figure 4.2 shows that typically fewer than 100 fragments account for 90% of the execution time. The lightly colored bars again represent programs that Dynamo would have chosen to bail out on. For example, in the benchmarks go and vortex, it is clear that a significant portion of time is not spent in a small section of the code. In the case of go, it took 1038 fragments to account for 90% of the execution time. The fragment cache is not large enough to handle such a large number of fragments and the generation of those fragments furthermore carries too much overhead. In such cases it is not beneficial to execute under Dynamo control, therefore the decision to bail out was a correct one.

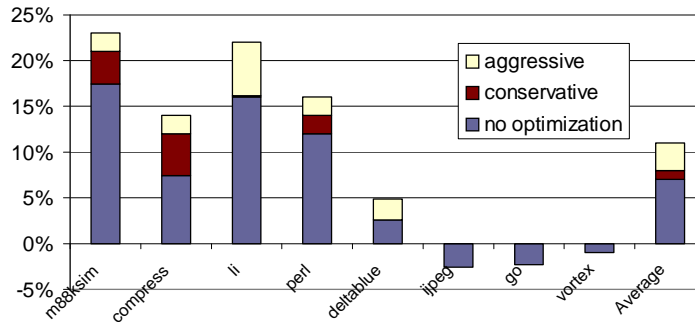


Figure 4.3: Wall Clock Time Comparison of Dynamo Performance to Native Execution.

A notable feature of the results obtained by the Dynamo group is that effort was made to produce meaningful results. This was done by comparing the wall clock time of programs employing Dynamo versus programs run directly on the hardware. Figure 4.3 shows the wall clock comparison of runtime of each of the benchmarks as compared to its native execution using the default level of optimization [6]. Only those programs where a bailout occurred resulted in a slowdown. Overall an average speedup of 11% was attainable over all of the benchmarks. This speedup was the result of aggressive optimization - those optimizations that do not guarantee precise exceptions. Performing only those optimizations that provided for precise exceptions resulted in an average of 8% speedup. Yet the most notable aspect of Figure 4.3 was the fact that an average speedup of 7% was achieved with absolutely no optimization, due to improved instruction cache locality.

4.3 A High-Level Description

Several factors led to the choice of ISA and operating systems for the new DO infrastructure. IA-64 is a prime ISA because there are many optimization opportunities inherent in an architecture that depends exclusively on static scheduling. Providing an infrastructure that supports the study of new dynamic optimization algorithms tailored for IA-64 could be quite beneficial to the researchers and users of IA-64.

Linux is a natural choice of operating systems in a research environment because the source code is publicly available, if necessary. Once correctness and performance has been proven, it is possible that a research version of dynamic optimization could be incorporated into future versions of Linux. It is far less likely that a solution such as DO would appear in the operating system owned by a large corporation, due to the "not invented here" mentality. Furthermore, because Linux is open-sourced, this dynamic optimization solution will be open to the suggestions and collaboration of thousands of researchers throughout the world.

Microsoft's Windows operating system is an interesting platform for the study of dynamic optimization because of its idiosyncrasies when compared to UNIX operating systems. Recent research [29] indicates that the dominant execution paths in interactive software benchmarks, such as the WindowsNT benchmarks, may be less predictable than

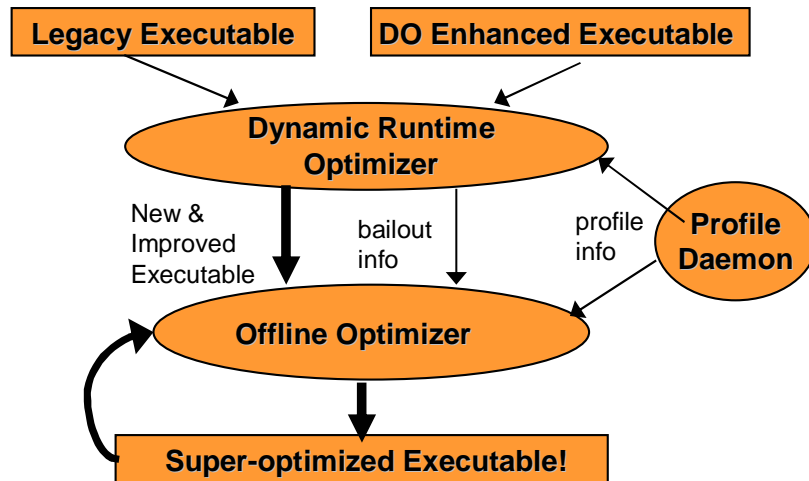


Figure 4.4: The DO Dynamic Optimization Infrastructure.

the SpecInt95 benchmarks due to the large program dependence on user interaction. It could be the case that an entirely new set of optimization rules should be employed for interactive software. This notion could be studied using the DO infrastructure for Windows.

The next notable extension from Dynamo incorporated into the new DO infrastructure is an offline optimizer. The offline optimizer will use information obtained during online optimization to perform any optimizations that were deemed too time consuming to be performed during runtime, such as complete program rescheduling. While the notion of offline optimization in itself is not new, there has yet to be an optimization infrastructure that features both online and offline optimization in a complementary manner.

In addition, various minor extensions will be incorporated in order to set DO apart from its Dynamo predecessor. One such extension is the periodic reevaluation of program fragments that are resident in the fragment cache. Limited studies have been performed to detect the existence of *profile shifts* [36] also known as *phased behavior* [37]. While the studies were not extensive enough to be conclusive, it makes intuitive sense to verify that an optimized fragment that is resident in the fragment cache should still be considered optimized. A second minor extension will be the consideration of optionally including compiler generated program annotation to aid in dynamic optimization. While this annotation may turn out to be useful for speeding up runtime optimizations, permanently requiring such annotations will not be considered due to the goal of backward compatibility with existing binaries. Another extension corrects for one of the weaknesses noted in the Dynamo description (Section 2.1.1) concerning Dynamo bailout. As stated before, the current Dynamo model could theoretically be initiated and fail for every instance of a program due to the bailout mechanism. To remedy this situation, DO will maintain information regarding the reason for each bailout beyond the execution time of a particular program. This will reduce the number of repeated attempts of the identical optimization in subsequent runs. A final extension will be the study of incorporating DO into the operating system. This concept is interesting because it allows for a level of control not possible in user-level software.

Currently, this design idea is only feasible in the Linux OS as the source code is publicly available and can be altered relatively easily.

Figure 4.4 introduces the DO infrastructure. The inputs of the dynamic optimization system can be either (a) existing legacy code compiled in a standard fashion, or (b) code compiled along with a DLL containing program annotation for the executable that can assist in later optimization decisions.

At runtime, the dynamic runtime optimizer and the profile daemon are deployed. The profile daemon is software resident and works closely with the interpreter, but unlike in Dynamo, hardware profile counters are available to it. The profiling counters present in the IA-64 processor are incremented for selected events such as branch mispredictions. The profile daemon manages the hardware counters (along with software-based counters, if more are needed) and notifies the dynamic optimizer when any hot spots are detected. The dynamic runtime optimizer (1) interprets the data provided by the profile daemon regarding potential optimizations that may be performed (2) performs cost-benefit analysis for the optimization and (3) performs particular optimizations if feasible.

Next, the dynamic runtime optimizer sends information to the offline optimizer. This information consists of (1) data regarding runtime optimizations that were performed, (2) data regarding optimizations that were not feasible at runtime, but could be performed offline, and (3) data regarding optimizations that were attempted at runtime, but were aborted due to performance constraints.

During the idle time of the computer, heavyweight, time-consuming optimizations are performed by the offline optimizer using data from the runtime optimizer. Because the offline optimizations are performed during idle computing time, no performance constraints exist. Furthermore, offline optimizations are permanent so they need only be performed when there is a change in program usage patterns or a processor upgrade.

The Dynamic Optimization systems presented in Sections 2.2.1 - 2.2.3 each perform some of the tasks presented in this section, but none of the systems perform all of the tasks. The DO infrastructure is intended to be a complete, realistic extension of Dynamo. It removes the constraints of other research infrastructures. Software developers do not need to learn new languages, use special compilers, or use special compiling flags. Users do not need to know or care about the dynamic optimization system in place; they do not need to initiate the optimization process or worry whether or not their software is compliant with the dynamic optimization system. The DO dynamic optimization model is completely transparent to the user. It combines lightweight online optimization techniques with heavyweight offline techniques to produce an effective, complementary optimization system.

Chapter 5

Dynamic Optimization Opportunities and Algorithms

A challenging aspect of dynamic optimization is deciding if and when to optimize a portion of code. Dynamic optimizations should be performed as soon as the profile information is available, on the other hand, the profile information should reflect overall program behavior. To further complicate the decision-making process, time should not be wasted optimizing rarely executed code. Instead, hot spots [6] should be located in order to focus on optimizing active portions of code. These uncertainties warrant a study of the most favorable optimization opportunities.

Many of the optimization algorithms presented in this chapter rely on profile information gathered during a particular run. For example, the dynamic if-conversion algorithm (see Section 5.2.1) relies heavily on the misprediction rate of the branches during the current execution. Not only is it important to gather accurate branch misprediction rates, but the profile data must be gathered early enough for the program to benefit from dynamic if-conversion. Therefore a study of several sampling heuristics is warranted before going into detail about new optimization algorithms.

5.1 Sampling Study: Branch Mispredictions

Because dynamic profiling might not be free (as in the case of Dynamo), the misprediction rate of a branch may be sampled and dynamic optimization decisions may be based on that sample rate. Therefore, an important question is: *How representative of the overall misprediction rate is a sample misprediction rate?* The answer to this question varies depending on the heuristic used to collect the statistics. This section takes a close look at several heuristics used to collect misprediction rate statistics for use in dynamic optimizations.

5.1.1 Misprediction Sampling Based on First N Occurrences

One scheme for gathering sample misprediction data is fairly straightforward. The misprediction rate for the first n occurrences of a branch is tracked, then it is assumed that

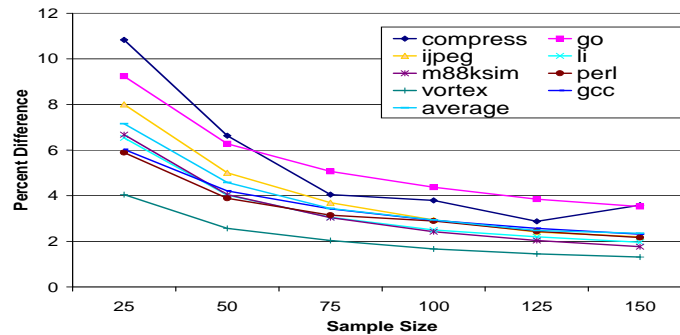


Figure 5.1: The Average Difference between Sample (First N Branch Occurrences) and Actual Misprediction Rates for all Branches in the SPECint95 Benchmarks.

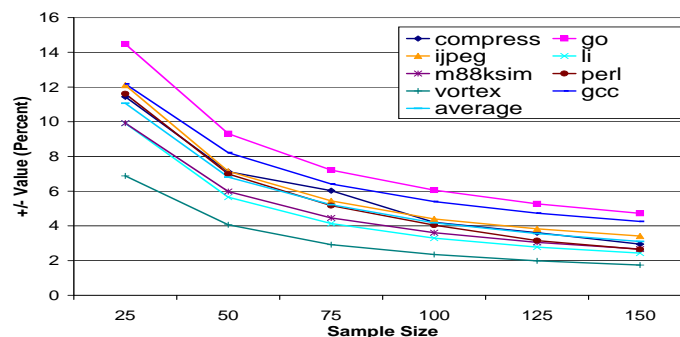


Figure 5.2: Error Values for a 99% Confidence Interval.

the behavior of that branch will follow the same trend in the future. Figure 5.1 shows the percent difference between sampled and full branch misprediction rates, averaged over all branches for the SPECint95 benchmarks.

The results in Figure 5.1 were calculated by determining the correlation of the misprediction rate of various sample sizes (25, 50, 75, 100, 125, 150) to the full-run misprediction rate, averaged over all static branches. The sample sequences are successive outcomes of branches taken at the beginning of program execution. This process is repeated for all branches in a benchmark. The outcomes are averaged in order to produce a single graph line for each benchmark.

The rates were viewed as a set of Bernoulli trials [16] where Bernoulli distributions were determined. A Bernoulli distribution is a means for representing a trial with two possible outcomes, in our case a misprediction or a correct prediction. Next, the two distributions - the overall outcome and the sample outcome, were compared using the Kolmogrov-Smirnov test [3]. A 99% confidence interval was chosen to determine the difference between the two distributions.

Figure 5.1 shows that the average difference in the misprediction rate of the SPECint95 benchmarks for a sample size of 25 branch occurrences when compared to the

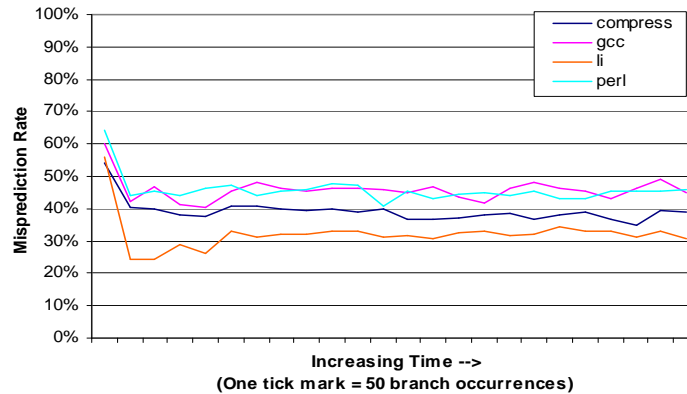


Figure 5.3: Misprediction Rate Over Time for Certain Hard-to-Predict Branches in SPECint95.

overall behavior of the branch is around 7%. This would equate to a misprediction rate of 17% in the sample run as compared to a 10% misprediction rate overall. If the sample size were increased to 150 branch occurrences, the average difference drops down to around 2%.

An important factor to keep in mind is that the average differences shown in Figure 5.1 themselves have a delta error value caused by the averaging. These error values are shown in Figure 5.2. The error values in Figure 5.2 encompass a 99% confidence interval. For example, we are 99% certain that the difference between the 25-sample misprediction rate of the vortex benchmark and the actual rate is 4% +/- 7%, or between -3% and 11%. Figure 5.2 allows us to calculate the worst-case scenario that can be expected when comparing sample misprediction rates to actual rates for any given branch.

An important conclusion that can be drawn from Figures 5.1 and 5.2 is that choosing the first n branch occurrences as the sample for estimating overall branch misprediction rate can be quite inaccurate for small values of n such as 25 or 50. In the worse case, inaccuracies of up to 26% may be seen. Sample sizes of 75 or greater result in error values of less than 4% on average with an error of less than 4% (or between 0% and 8%). Yet as the sample size is increased, the benefits gained from dynamic optimizations are reduced because the point at which the optimizations are performed is delayed.

5.1.2 Adaptive Warmup Exclusion

Another approach for sampling misprediction rates involves recognizing an end-of-warm-up (confidence) condition and only then beginning to collect misprediction statistics. Section 5.1.1 described a sampling heuristic that compared the first n outcomes of a particular branch to its overall outcome statistics for the entire program duration. The problem with this approach is that the data was gathered at the start of the program. The start of any program typically has a high misprediction rate as the branch predictors learn the branch patterns. This behavior may not be representative of the entire program behavior because the misprediction rate of a particular branch often decreases notably after the

$$|P_{MISSA} - P_{MISSB}| < T$$

P_{MISSA} = last misprediction rate

P_{MISSB} = this misprediction rate

T = threshold

Figure 5.4: Determining the End-of-Warmup Condition.

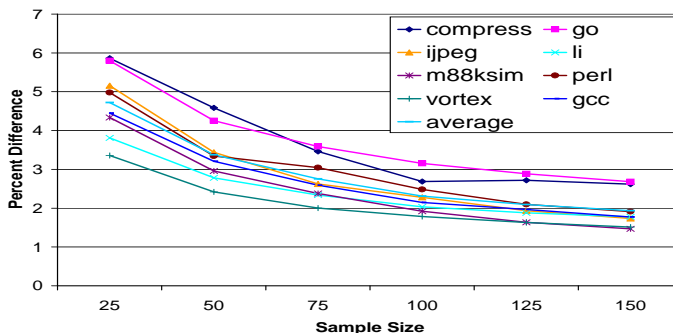


Figure 5.5: Mean Difference between Sample (First N Branch Occurrences Ignoring Warm-up Period) and Actual Misprediction Rates.

warm-up period, as shown in Figure 5.3 for several critical and/or hard-to-predict branches in SPECint95. Several conclusions can be drawn from Figure 5.3. First, it is apparent that the behavior at program start is an inaccurate indicator of overall behavior. Second, branch misprediction rates typically do not vary by more than 10% as the branch continues execution.

The *adaptive warmup exclusion* sampling heuristic attempts to recognize the stabilization of the branch predictors on a branch-by-branch basis. Only when a branch reaches the end of its warmup period does the profiling of misprediction statistics for the branch begin. The algorithm for detecting an end-of-warmup condition is shown in Figure 5.4. The algorithm attempts to recognize an end of warmup condition as the point at which the branch misprediction rate settles to within a threshold value (10% for example) of the previous rate. The results of testing this heuristic with a threshold value of 10% are shown in Figure 5.5. The graph shows us that by ignoring the warm-up period of the branch predictors, average sampling accuracy improves by 43% (dropping from 7% difference to 4% difference).

Figure 5.6 shows the effect of holding the sample size at a single value and varying the threshold value from 10% to 5% 1% and 0.5%¹. The threshold is defined as the difference in misprediction rates of two subsequent branch occurrences. As the graph indicates, varying the threshold value results in minimal changes to the mean difference between

¹To avoid false positives, a misprediction rate change of 0% does not trigger an end of warm-up condition. Therefore a rate change from 100% to 100% after two samples would not prematurely end the warm-up period.

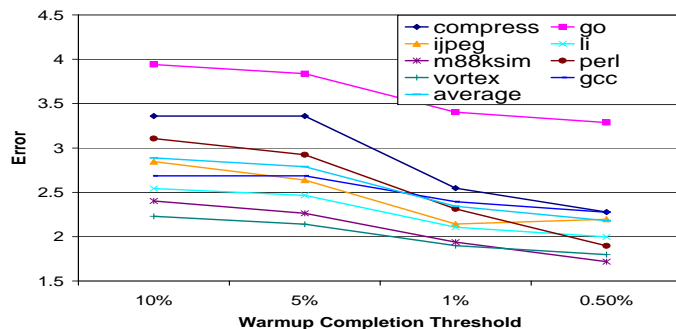


Figure 5.6: Effect of Varying the Warm-up Completion Threshold Value.

the sample and actual misprediction rate. This study of the dynamic sampling heuristics provides insight into the effectiveness and accuracy of dynamic optimizations that are performed based on sampled statistics. If the sampling heuristic is inaccurate, any dynamic optimizations that make transformations based on sample data are likely to be inaccurate or unnecessary and may hurt overall program performance.

5.2 New Dynamic Optimization Algorithms

While performance features of the IA-64 architecture result in reduced run times and better hardware utilization, these features can also be uniquely leveraged by dynamic optimizing algorithms.

Dynamic optimizations may soon become a necessity for the scalability of the EPIC architecture. Because all scheduling is performed at static compile time, any program compiled for the Itanium chip will be optimized for execution on the Itanium. Such programs may not execute in the most effective manner on any subsequent chip. New function units and layouts will not be utilized since the instructions were scheduled for execution on a chip that did not possess such features. Dynamic rescheduling could eventually become a necessity in order to utilize the full potential of subsequent EPIC microprocessors.

A notable amount of scheduling flexibility is lost when moving from an out-of-order execution scheme to an in-order scheme as will be done in the transition from the Pentium (a superscalar) to the Itanium (an EPIC) chip. The Pentium chip line could manage a smooth transition from one chip design to its successor because it contained a reorder buffer that would issue instructions out-of-order. Essentially, the instruction schedule developed by the compiler was not set in stone; it could be substantially altered via dynamic scheduling.

This is not the case with the EPIC architecture. The schedule produced by the compiler is set in stone. No reordering of instructions is allowed because all dependency-checking hardware has been removed. While this compile-time scheduling scheme originally produces faster code, the speed of the post-compilation executable is not scalable to more advanced EPIC processors that will be released. For this reason, dynamically rescheduling executables may eventually become a necessity for scalable performance.

Furthermore, an extensive amount of dynamic optimizations can be performed that

will instantly improve the performance of an IA-64 executable on even the first processor.

This section takes a close look at some optimizations that may be performed at runtime. The results are obtained using an EPIC-style execution-driven simulator operating on the SpecInt95 benchmarks. Our LEGO back-end compiler [1], which is based on the HPL PlayDoh Architecture [24], was used to schedule the benchmarks. Tregion scheduling was used [20], combined with static if-conversion and most modern optimizations that would be present in an EPIC compiler such as aggressive list scheduling, speculative scheduling, liveness analysis, and support for multi-way branching. The branch predictor modeled is a hybrid predictor containing a PAS predictor and a Gshare predictor [15, 40].

One goal of modern compiler research has been achieving a synergy between static and dynamic optimizations. It is often the case that certain static optimizations eliminate opportunities for more effective dynamic optimizations, and vice versa. The run-time optimizations presented in this thesis are independent of the optimizations performed at compile time, and can therefore be applied to any code developed for the EPIC architecture regardless of the optimization level performed at compile time.

Predicate profiling and prediction is a concept similar to branch profiling and prediction. Because IA-64 compilers often choose predication over branches, predicting the value of predicates can allow instructions to be scheduled in an effective manner.

False predicate path collapsing is a concept that focuses on the weaknesses inherent in a predicated architecture, such as IA-64. This optimization looks ahead in the executing code and locates all instructions that are dependent on a predicate that has already been deemed false. These instructions are squashed from the instruction path and are never issued to the function units. While reducing the code size, this scheme potentially reduces power.

Dynamic rescheduling improves the scalability of the IA-64 architecture by providing the functionality to reschedule and reoptimize an executable. This functionality is necessary for improved performance on subsequent processors, as the processor may feature a different number or type of function units or pipeline structure.

Predicate promotion is an idea that allows instructions to break instruction group dependencies. This allows rescheduling to be performed between instruction groups, such that an instruction contained within one instruction group can be promoted to a previous instruction group and fill in any no-op slots available. Predicate promotion is performed using a new double predication concept, which will later be discussed in detail.

Finally, *dynamic instruction hint adjustment* builds upon the fact that IA-64 allows path-predictions to be hard coded into a branch instruction at compile time. Because such branch predictions are not based on run time behavior, a method for changing such predictions based on actual behavior should be incorporated.

Each of these optimizations will be discussed in detail in the following subsections.

5.2.1 Dynamic If-Conversion

If-conversion replaces a branch and its control dependent paths with guarded (or predicated) execution [4, 35]. Rather than evaluating a branch condition and conditionally executing only one control dependent path, both paths are executed and predicates

Before: if (cond) Branch L1 r2 = MEM[A] r1 = r2 + 1 r2 = MEM[r1] L1: r9 = r3 + r4	After: p1, p2' = cond (p2) r2 = MEM[A] (p2) r1 = r2 + 1 (p2) r0 = MEM[r1] L1: r9 = r3 + r4
--	---

Figure 5.7: If-Conversion Example.

control which results are used. It is a means for converting control dependences into data dependences.

Much of the current research involving if-conversion is based on the work done at Hewlett-Packard Laboratories [35]. A basic if-conversion algorithm is presented, called the RK algorithm. The R function specifies which instructions should be based on the same predicate value, while the K function indicates the conditions under which a predicate should be set to true.

If-conversion avoids the penalty incurred when a branch is otherwise mispredicted. It is not a good idea to if-convert all branches, however. In certain instances, it is better to branch over a large set of instructions rather than execute all of them. If-conversion works best when attempts are made to reach a balance between control flow and predication [5]. Yet a perfect balance is not attainable at static compile time. The disadvantage is that if-conversion is performed at static compile time when little information is known about the behavior of a particular branch.

Mahlke et al. [31] introduced compiler support for predicated execution using the hyperblock. The hyperblock is used to increase the scheduling scope. This is accomplished by allowing speculative execution of predicated instructions. In addition, basic blocks based can be selectively included based on frequency and size. Mahlke then studied the impact of predicated execution on branch prediction and discovered that 56% of dynamic branch mispredictions are eliminated with predication support [33]. Our work may be viewed as an extension of the work done by Mahlke into the dynamic domain.

Aside from Mahlke, several other researchers have delved into if-conversion. One example is the work of Klauser et al. [27]. They describe a method for dynamically introducing predication to architectures that do not already support it. While their main objective was to provide hardware support for the dynamic introduction of predication into programs, our main goal is to use software to enhance performance on an ISA that already supports predication.

Dynamic if-conversion is the runtime equivalent of static if-conversion. It can be used to if-convert branches that are performing poorly during a particular run. It can also be used to ensure that if-conversion is performed at all, as some binaries may not have been statically compiled with if-conversion.

The decision between branching and predication should take into account (1) branch misprediction penalty (2) misprediction rate (3) fetch/execution overhead for otherwise correctly predicted branches. While the misprediction penalty may be known at

```

if (x) Path A
else Path B

```

$$L_A + L_B \leq L_A * P_A + L_B * P_B + L_{Br}$$

L_A = latency of Path A
 L_B = latency of Path B
 P_A = probability of Path A
 P_B = probability of Path B
 L_{Br} = latency of a branch

Figure 5.8: A Predication Algorithm.

$$\sum L_i \leq \sum ((L_i + L_{Br}) * P_i)$$

Figure 5.9: A General Predication Algorithm.

compile time, such a penalty may change over time due to processor upgrades. It can be claimed that misprediction rate can never be accurately determined before runtime. While branch profile information from previous executions may be available, there is no guarantee that such information is still relevant.

Simply predicating all paths of execution is certainly not the best approach when many of the paths in question have a very high likelihood of not requiring execution. In fact, predication is the best approach only if both of the following assumptions are true: (1) each path in the original branch contains a small² number of instructions, and (2) each path has a relatively equal chance of being executed. In other words, it is only worthwhile to predicate instructions if it is cheaper to execute all paths than to branch over the untaken path (and risk a misprediction).

Figure 5.8 introduces an equation for deciding whether or not to predicate instructions. This algorithm states that predication is the recommended method of execution if the latency of executing both paths is less than the latency of executing one path plus a branch instruction. It is important to include the corresponding potential pipeline stalls due to misprediction into the latency of the branch instruction. Figure 5.9 is a generalized version of Figure 5.8 that can be applicable to multiple path execution decisions, such as a case statement.

As IA-64 processors evolve, the predication vs. branching calculations may change. Reducing the latency of certain operations may result in branches that should now be converted to predicated instructions for good performance, or vice versa. Reevaluation of predication should occur with each new implementation. Dynamic optimization effectively provides this flexibility.

The Dynamic If-Conversion Adjustment Algorithm. Deciding whether or not to

²In this case, small is defined as the number of penalty cycles incurred for taking the branch, it is not worthwhile to predicate the instructions.

$$P_{MISS} - L_{MISS} \geq P_{HIT} * L_{HIT} * (1 + error)$$

P_{MISS} = misprediction rate

L_{MISS} = misprediction penalty

P_{HIT} = correct prediction rate

L_{HIT} = cycles to execute predicated instructions

Figure 5.10: Dynamic If-Conversion Algorithm.

$$A_T - A_B > 0$$

$$A_T - A_B < L_{MISS} * P_{MISS} * S_{INSTR}$$

A_T = target address

A_B = branch address

L_{MISS} = misprediction penalty

P_{MISS} = misprediction rate

S_{INSTR} = instruction size

Figure 5.11: Maximum Allowable Branch Distance.

convert a branch to a set of predicates at runtime is a matter of comparing the average latency of either option.

First of all, the branch must be a forward branch, that is, the target address must be greater than the branch address. This restriction is put in place because backward loops typically do not benefit from branch to predicate conversion. Not only are loops highly predictable, but they always require a branch before the epilogue. This branch cannot be converted to predication. In the case of loop unrolling, the unrolled portions of a loop that end up benefiting from branch conversion become forward branches in the unrolling process.

The second requirement is that a candidate branch must be on a hot path. The cost of converting a branch to predicated code is typically amortized over the subsequent uses of the section of code. Therefore, time and effort should not be wasted on a branch that is not likely to be encountered frequently.

Figure 5.10 describes the dynamic forward if-conversion (branch-to-predicate) adjustment decision. The left side of the equation estimates the average penalty for leaving the branch as is. The right side estimates the overall cost of removing the branch and predicating the following set of instructions. The error value mentioned in the right side of the equation compensates for the error involved in determining the branch misprediction rate. Since only a sample of the misprediction rate has been gathered when Figure 5.10 is evaluated, the error term compensates for the error with a 99% confidence interval.

Branches over very large sections of code will typically not be considered for forward if-conversion because the cost of executing all of the predicated instructions on the off path far outweighs the benefits of removing the branch misprediction penalty. In the case of IA-64, where the branch mispredictions penalty is 10 cycles [23], most of the converted branches will have a target address that is within 10 cycles of the branch instruction itself.

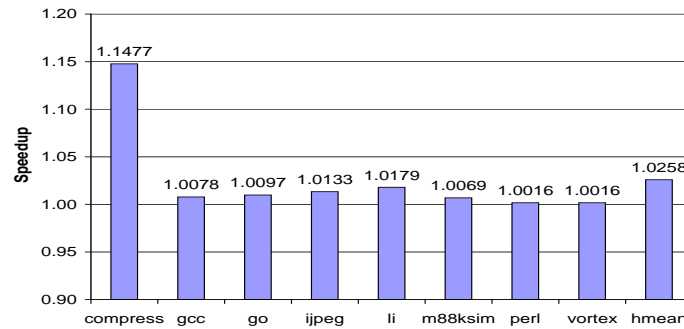


Figure 5.12: Speedup from Applying the Dynamic If-Conversion Algorithm.

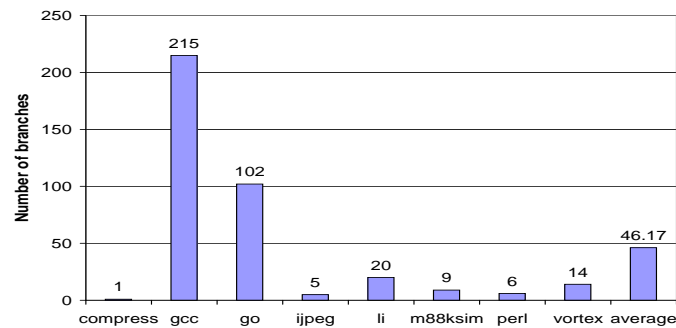


Figure 5.13: Number of Branches Converted to Predicates.

Higher distances are permitted in the case of unusually high misprediction rates. The maximum value of the allowable branch distance (target address - branch address) is roughly the misprediction penalty in cycles multiplied by the misprediction rate multiplied by the instruction size as shown in Figure 5.11.

The basic concept is that the latency of a few predicated instructions may be much smaller than the latency of a potential branch misprediction. And if a branch misprediction is likely to occur, it is better to convert that branch to predicated instructions. The potential for a branch misprediction is then eliminated and, often, the predicated instructions can be scheduled into the holes (NOPs) of the existing schedule.

Dynamic If-Conversion Results. The effectiveness of the dynamic branch adjustment algorithm was determined by comparing simulated run times on the SPECint95 benchmarks on an IA-64 processor employing a Yeh-Patt PAS/Gshare Hybrid branch predictor. Although the actual Itanium branch predictor has not been announced as of this writing, the Yeh-Patt PAS/Gshare Hybrid branch predictor is one of the most effective predictors available to date.

Figures 5.12 and 5.13 show the speedup of applying dynamic if-conversion and the actual number of static branches converted to predicates for the SPECint95 benchmarks. There was a great deal of variance in the number of branches converted (from 1 to 215)

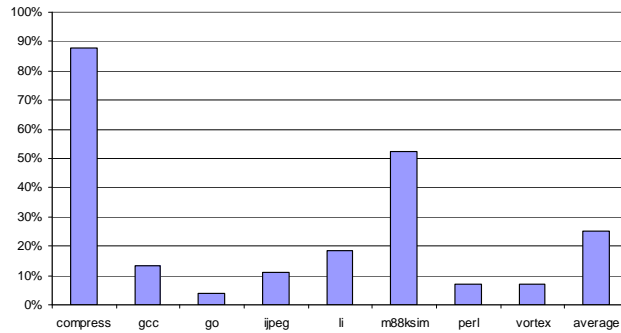


Figure 5.14: Percent of Branch Misprediction Eliminated using Dynamic If-Conversion.

and the corresponding speedup (from 0.16% to 14.7%), yet no slowdown was detected. On average, 46 branches were converted to predicates for a speedup of 2.5%. Since 95% of branches are predicted correctly using the Yeh-Patt/Gshare hybrid branch predictor, this algorithm works to improve the 5% of mispredicted branches.

Figure 5.14 shows that on average, 25% percent of branch mispredictions are eliminated when the dynamic if-conversion algorithm is in use. From these results, the dynamic branch adjustment algorithm complements the branch predictor, effectively eliminating much of the 10-cycle branch misprediction penalty in Itanium.

5.2.2 Dynamic Reverse If-Conversion

Dynamic if-conversion predicates some branches that were not predicted by the static compiler; the opposite scenario could also be true. The compiler may have converted certain basic blocks to predicated regions yet the decision may be more valid for some runs than others. The predicates may turn out to be quite biased for a given run. In this instance, it would be better if the predicated region was converted back into a branch.

Dynamic predicate-to-branch conversion, or *dynamic reverse if-conversion*, converts predicated regions back into control-flow regions at runtime. The idea is that the predicates guarding the instructions evaluate true so rarely that it would be better to branch over the instructions rather than to predicate them.

The Dynamic Reverse If-Conversion Algorithm. The algorithm for dynamically converting predicates to branches is shown below. It is very similar to the forward if-conversion algorithm in that it weighs the cost of each option - leaving the instructions as predicated instructions or converting them to branches.

The left side of the equation calculates the penalty for leaving the instructions as a set of predicated instructions, while the right side predicts the penalty for converting the predicates to a branch. The odds of misprediction are not known for a set of predicates because the branch predictor is not used for predicated instructions. Therefore, the average misprediction rate for the current branch predictor is used.

Dynamic Reverse If-Conversion Results. Figures 5.16 and 5.17 show the speedup of the dynamic reverse if-conversion algorithm for SPECint95 and the number of predicate

$$P_{PRED'} * L_{PRED} \geq P_{MISS} * L_{MISS}$$

$P_{PRED'}$ = odds of false predicate

L_{PRED} = number of predicated cycles

P_{MISS} = odds of mispredict

L_{MISS} = misprediction penalty

Figure 5.15: The Dynamic Reverse If-Conversion Algorithm.

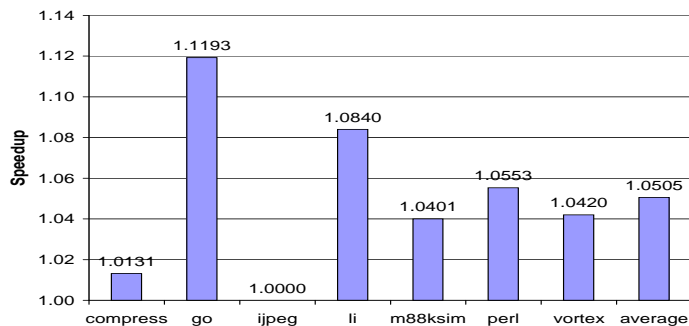


Figure 5.16: Speedup Resulting from Dynamic Reverse If-Conversion.

traces converted to branches for each benchmark, respectfully. The graphs show an average speedup of 5% for converting an average of 27 predicate traces back into branches. The speedup in Figure 5.16 results from removing relatively long sequences of instructions that had been guarded by a typically-false predicate. Rather than endure the performance penalty of continually executing instructions predicated on a false predicate, the predication was converted to control flow, and a branch was placed before the instructions.

Implementation Details. Because the dynamic if-conversion algorithms are lightweight, they can be easily implemented in a modern dynamic optimization system with low overhead. Since many dynamic optimizers already perform some sort of hot-path detection, we can be assured that time is not be wasted performing optimizations on rarely executed instructions.

While the algorithms require some sort of profiling structure, it is not limited to a pure hardware or software solution. For example, Itanium provides built-in performance monitors [23] and Dynamo uses software structures such the branch and trace counters [6]. Dynamo already associates a software counter with many branches and these counters could also be leveraged to determine candidate branches for predicate conversion. In addition, Dynamo also maintains a fragment cache containing recently optimized fragments (executable traces).

Furthermore, incorporating the dynamic if-conversion algorithms can improve the performance of other algorithms already in place. For example, when combining dynamic if-conversion with aggressive dynamic rescheduling, the performance can potentially be better than additive. Dynamic if-conversion creates many more optimization opportunities

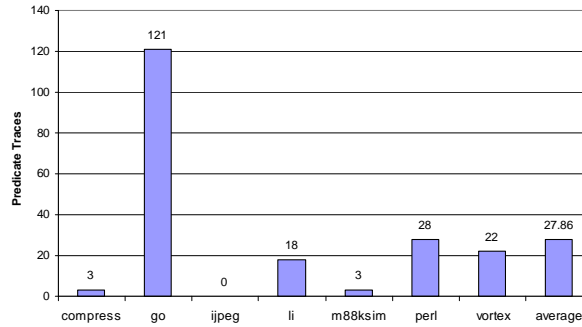


Figure 5.17: Actual Number of Predicate Traces Converted to Branches.

for rescheduling because there is much more flexibility in scheduling predicated instructions than control-flow sequences. This, combined with the fact that the algorithms are not tied to any particular implementation, make the dynamic algorithms an effective option for any system.

5.2.3 Dynamic Rescheduling for Scalability

Superscalar processors often supports out-of-order execution, i.e. dynamic scheduling. In the EPIC architecture, all scheduling is performed at compile time. The compiler must have knowledge about the number and type of function units in advance in order to schedule instructions correctly. What happens when another EPIC processor is released that contains more and/or differently configured function units than its predecessor? All code must be recompiled to utilize the additional function units; otherwise the new units will lie idle. How can an ISA such as IA-64 survive processor upgrades? A process known as *dynamic rescheduling* [11] could be the answer.

Dynamic rescheduling is intended to be a one-time program overhaul, where instructions are broken down into instruction groups and instruction bundles are reformed to reflect the new hardware. Dynamic rescheduling need only be performed once for each new processor encountered.

Figure 5.18 shows the dynamic rescheduler converting an executable optimized for the first Itanium chip into an executable optimized for the next-generation Itanium chip.

While it may be possible to do a limited amount of rescheduling online, performing an entire program overhaul may be best left as an offline process. If performed offline, a greater amount of optimization can be performed including heavyweight optimizations, such as register reallocation, without concern for the time it takes to perform such optimizations.

Another use for dynamic rescheduling is as an optimization tool when a profile shift [36] or phased behavior [37] is detected. This form of dynamic rescheduling is necessarily an online process that occurs each time a substantial profile drift is detected. Several of the dynamic optimizations presented in the next subsections can be quite useful for an online version of dynamic rescheduling.

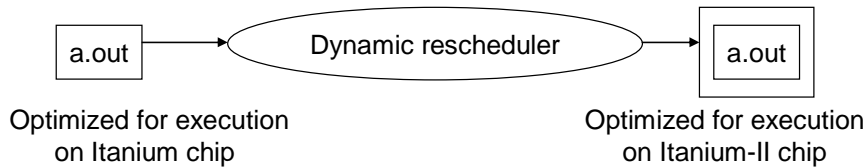


Figure 5.18: Dynamic Rescheduling.

```
(p3)  add r1 = r2, r3;;
      ...
      (p5)  add r8 = r1, r2
```

Figure 5.19: IA-64 Instructions with an Instruction Group Boundary.

5.2.4 Predicate Promotion with Double Predication

Currently, stop bits indicate a dependence boundary that cannot be broken in the IA-64 instruction sequence. The stop bit signifies a synchronization point, or instruction group boundary, where all instruction preceding the stop bit must be executed before moving on to any other instructions. Reordering instructions within an instruction group is rather straightforward, granted the templates for instruction bundles are followed. Since no dependencies or hazards are allowed within a single instruction group, the order of instructions can be altered as much as the templates allow. Rescheduling only within instruction groups is too restrictive, however. An algorithm for breaking the instruction group dependence boundary could be quite useful for extending dynamic rescheduling.

Predicate promotion is one method for breaking the instruction group boundary. In Figure 5.19, the instructions following the assignment of `r1` cannot be promoted above the instruction group boundary because of a possible dependence on `r1`. It may be the case, however, that `p3` turns out to be false, and no dependency exists. Using an extra predicate, `p7` - evaluated as `p5 && !p3` - the instruction can be promoted above the instruction group boundary. It can then fill in a no-op slot in a previous instruction group. This concept is known as double predication, and it allows instructions to be promoted above instructions that possibly have a dependency. Of course, this promotion is only useful if there are NOP slots to be filled in a previous instruction group.

Figure 5.20 demonstrates predicate promotion with double predication. In the

```
      cmp.add p7, p8 = p3, p4
(p7)  add r8          = r1, r2
(p3)  add r1          = r2, r3;;
(p8)  add r8          = r1, r2
```

Figure 5.20: Breaking the IA-64 Instruction Group Boundary.

Before: (p4) add r1 = r2, r3 (p4) add r1 = r1, r4 (p5) mov r2, r3 (p5) add r2 = r3, r1 (p5) sub r2 = r2, r1 (p5) br r1	\implies	After: (p5) exit fragment (p4) add r1 = r2, r3 (p4) add r1 = r1, r4
--	------------	--

Figure 5.21: An Example of False Predicate Path Collapse.

case that there is not a RAW hazard with `r1`, the latency of the following `add` instruction is hidden by promoting the instruction into a NOP slot of a preceding instruction group. While there is a one-instruction overhead (the `cmp` instruction), this instruction can typically fit in a nearby NOP slot in the schedule.

5.2.5 False Predicate Path Collapse

While converting branches to predicated instructions reduces pipeline bubbles, it also means that many instructions are executed that are not on the taken path. Because all potential paths through an executable must be scheduled at static compile time, a great deal of the scheduled code is unnecessarily executed but not committed due to dependence on a false predicate. Such cycles could be better spent executing real instructions.

The results of predicates are not known at static compile time. They are, however, known at run time. It is at run time that this knowledge can be very useful. The concept of false predicate path collapse refers to the immediate, albeit temporary, removal of all instructions whose predicate evaluates false. Removing unnecessary instructions from the scheduled instruction stream will result in a better instruction sequence being issued to the processor.

In the case of a cached supply of optimized instructions, false predicate path collapse can be applied and instructions based on a false predicate can be removed from the cached set of instructions. Any corresponding change of the predicate value would then invalidate the cached instructions associated with that predicate (see Figure 5.21). This concept differs from reverse if-conversion in that rather than inserting a branch over the set of instructions, the instructions are actually removed from the cached instruction stream.

5.2.6 Predicate Profiling and Prediction

At static compile time, no information is known about the typical behavior of instruction predicates. Currently, all IA-64 instructions are scheduled with no knowledge of predicate behavior. Regardless of how many times an IA-64 instruction sequence has been executed, no changes are made to the original schedule to reflect the behavioral pattern of predicates or branches.

Maintaining profile information for predicates could aid in many ways, including better scheduling techniques that could in turn be used with false predicate path collapse.

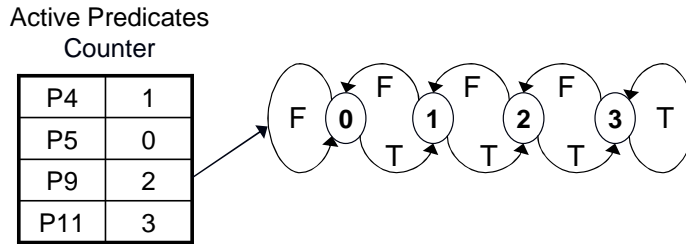


Figure 5.22: Predicate Profiling Mechanism.

Branch Completer	Branch Hint
spnt	Static Predicted Not Taken
sptk	Static Predicted Taken
dpnt	Dynamic Predicted Not Taken
dptk	Dynamic Predicted Taken

Table 5.1: IA-64 branch instruction completers.

If instructions that are dependent on predicates that are frequently false are moved to the end of an instruction group, they could be easily squashed from the instruction stream with no delays in execution time.

Furthermore, consider a sequence of instructions that are all dependent on a predicate that is evaluated as false 99% of the time. It would be more effective to speculatively remove these instructions from the instruction stream and provide a recovery mechanism for the 1% of the time that the instructions actually require execution.

A simple software profile table can be built containing information about the predicates that are used during a particular run as shown in Figure 5.22. Counters associated with the predicates can maintain information regarding the likelihood of the predicate evaluating to a certain value. Since the possible values are either true or false, the process of determining the likelihood of evaluation to one of the values is rather straightforward. A simple two-bit saturating counter is a lightweight method for determining information about the behavior of predicates that can therefore be used in dynamic rescheduling.

5.2.7 Dynamic Instruction Hint Adjustment

Branches are hard coded at compile time with branch extensions (called completers) that specify the static predicted outcome of the branch. Table 5.1 shows the allowed completers.

It is clear that a misjudgment at static compile time would result in suboptimal branch performance on every execution. Ideally, feedback from a program monitor would detect and permanently correct such mispredictions within the code. Yet this is not currently implemented.

Such an implementation would require minimal effort considering the monitoring

features that are already incorporated into the Itanium processor. The Itanium processor supports feedback-directed optimization by including special features such as monitoring registers in hardware. These low-overhead resources need only be tapped by a dynamic optimization system in order to produce the necessary means to support optimizations such as dynamic branch hint adjustment.

Chapter 6

Conclusions and Future Work

Given the compile-once-run-anytime nature of modern software distributions, software cannot be recompiled to achieve better performance on the latest processor. Dynamic optimization steps in and allows software to be reoptimized at runtime to perform well on any processor within an ISA. Dynamic optimization also provides a level of user-customized optimization not possible with traditional compilation models.

In this thesis, current approaches dynamic optimization, compilation and translation infrastructures are surveyed. The strengths and weaknesses of each model are studied to propose an effective infrastructure for dynamic optimization. The Dynamo dynamic optimizer from Hewlett-Packard Laboratories is recommended as a base model on which to build DO, a combination online/offline optimizer. The DO extensions include optional compiler-inserted annotations for reduced runtime optimization, an offline optimizer that is capable of performing heavyweight optimizations such as instruction rescheduling, and a mechanism for detecting profile shifts and recording information about optimization bailout. The DO infrastructure may be implemented in both the Linux and Windows environments in order to form a framework for studying runtime and post-runtime dynamic optimization algorithms.

Finally, two dynamic optimization algorithms are developed and evaluated in detail: dynamic if-conversion and dynamic reverse if-conversion. Determining an optimal means for tracking the misprediction rate of a branch and recognizing the need for dynamic if-conversion warranted a study into the matter. As it turns out, if the warm-up period for a branch (defined as the time it takes for the branch misprediction rate to settle to a value within 1-10% of the previous value) is ignored, the error of our sample misprediction rates drops from 7% to 4% for a 25-event sample. For larger sample sizes, the error drops below 2%. The accuracy of the sample misprediction rates is important because the rates form the basis of our dynamic if-conversion algorithms.

Dynamic if-conversion and dynamic reverse if-conversion was then introduced as a set of algorithms that take into account actual branch and predicate behavior to calculate the trade-offs of if-conversion at a particular runtime instance. The algorithms are lightweight - they can be implemented with minimal impact on system performance, and they are universal - they can be implemented in any dynamic optimization system available without the need for specialized hardware. By simulating the algorithms on an EPIC-style

machine employing the latest branch prediction scheme, speedup values of up to 14.7% were observed.

Various additional dynamic optimization algorithms were then presented including predicate profiling and prediction, dynamic rescheduling, predicate promotion, and dynamic instruction hint adjustment.

The specific optimizations presented in this thesis are useful in the domain of dynamic optimizations for the IA-64 architecture. Incorporating these algorithms into an environment for constant feedback-directed optimization of IA-64 executables could result in a major performance boost, while opening the doors for the future performance scalability of the EPIC architecture.

Furthermore, the release of a dynamic optimization infrastructure into the research world can ease the effort involved in studying and testing future algorithms developed worldwide. Dynamic optimization has a clear future that can only be enhanced by further research.

Dynamic optimization is a relatively new research area. Several researchers have mapped out the future of dynamic optimization [37, 26], stressing problems that must be solved for dynamic optimization to be widely used. Future work is described in this context.

Build the Online/Offline Dynamic Optimizer. While many research groups have chosen to build online dynamic optimizers, few have built offline optimizers and none have built a combination online/offline dynamic optimizer. Online/offline optimization may be very effective if the offline optimizer uses data gathered from the online optimizer. The DO proposal presented in 0 is quite feasible for implementation as part of a Ph.D. thesis. While undertaking such a large project is challenging, having the infrastructure available to study the synergy between online and offline optimizations could pay off in the long run. In addition, releasing this tool to the research domain could then propagate the interest in studying dynamic optimization algorithms, as the methodology for testing would be greatly simplified.

Processor Enhancements to Support Dynamic Optimization. While most current dynamic optimization systems are completely software based, it is likely that microprocessor designers would be willing to provide built-in hardware support to aid in dynamic optimization. The form of support that could be provided is an open research area. A cost-benefit analysis of hardware support for dynamic optimization is warranted.

Optional Dynamic Translation from IA-32. The ability to reoptimize executables to perform better for each iteration of EPIC chips is a powerful tool. This ability would be extremely useful for converting x86 executables to an equivalent EPIC executable, optimized for the processor on which it would be run.

While one goal of the IA-64 architecture is complete backward-compatibility with the x86 architecture (also referred to as IA-32), x86 instructions executing on an IA-64 processor may run much slower than a natively compiled binary. Therefore a permanent conversion and rescheduling of x86 instructions for an IA-64 processor would be very useful. This idea is worth exploring in further detail in the future.

Bibliography

- [1] The LEGO compiler. <http://www.tinker.ncsu.edu/LEGO>.
- [2] An application programming interface (API) for runtime code generation. <http://www.cs.umd.edu/projects/dyninstAPI/>, 2000.
- [3] A. O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Harcourt Brace Jovanovich, Boston, 2nd edition, 1990.
- [4] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.
- [5] David August, Wen-Mei Hwu, and Scott Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 13th IEEE/ACM International Symposium on Microarchitecture*, pages 92–103, 1997.
- [6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
- [7] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [8] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March-April 1998.
- [9] Charles Consel and Francois Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 145–156, 1996.
- [10] Thomas M. Conte, Menezes. Kishore N., and Mary Ann Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, 2–4 December 1996. ACM Press.

- [11] Thomas M. Conte and Sumedh W. Sathaye. Dynamic rescheduling: A technique for object code compatibility in VLIW architectures. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 208–218. ACM Press, 1995.
- [12] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.
- [13] Kemal Ebcioglu and Eric Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th International Symposium on Computer Architecture*, pages 26–37, June 1997.
- [14] Dawson R. Engler and Todd A Proebsting. Dcg: An efficient, retargetable dynamic code generator. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 263–273, October 1994.
- [15] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ISCA*, pages 3–11, 1996.
- [16] Larry Gonick and Woollcott Smith. *The Cartoon Guide to Statistics*. Harper Perennial, 1993.
- [17] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in dyc'. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.
- [18] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.
- [19] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
- [20] William A. Havanki, Sanjeev Banerjia, and Thomas M. Conte. Treeregion scheduling for wide issue processors. In *HPCA*, pages 266–276, 1998.
- [21] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.
- [22] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, February 1997.
- [23] Intel Corporation. *IA-64 Application Developer's Architecture Guide*. Intel Corporation, Santa Clara, CA, USA, May 1999.
- [24] Vinod Kathail, Michael Schlansker, and Bob R. Rau. Hpl playdoh architecture specification. Technical Report HPL-93-80, Hewlett Packard Laboratories, February 1994.

- [25] Thomas Kistler. Dynamic runtime optimization. Technical Report ICS-TR-96-54, University of California, Irvine, Department of Information and Computer Science, November 1996.
- [26] Alexander Klaiber. *The Technology behind Crusoe Processors*, January 2000.
- [27] Artur Klauser, Todd M. Austin, Dirk Grunwald, and Brad Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *IEEE PACT*, pages 278–285, 1998.
- [28] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.
- [29] Dennis C. Lee, Patrick Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on windows NT. In *ISCA*, pages 27–38, 1998.
- [30] M. Leone and R. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization, 1997.
- [31] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.
- [32] Scott Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
- [33] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, 1994.
- [34] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, pages 136–147, 1999.
- [35] J.C.H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Software Systems Laboratory, May 1991.
- [36] Sumedh Sathaye. *Evolutionary Compilation for Object Code Compatibility and Performance*. PhD thesis, North Carolina State University, 1998.
- [37] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, pages 1–11, 2000.
- [38] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

- [39] Mark C. Toburen. Power analysis and instruction scheduling for reduced di/dt in the execution core of high-performance microprocessors. Master's thesis, North Carolina State University, 1999.
- [40] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.
- [41] Catherine Xiaolan Zhang, Zheng Wang, Nicholas C. Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26, 1997.