Binary Redundancy Elimination

Author: Manel Fernández
Advisor: Roger Espasa

Thesis submitted in fulfillment of
the requirements for the degree of
Doctor en Informática

UPC

UNIVERSITAT POLITÈCNICA DE CATALUNYA
Department of Computer Architecture
Barcelona, Spain

January 2005

A mis padres y hermanos.
Y muy especialmente, a Laura.

Abstract

Two of the most important performance limiters in today's processor families comes from solving the *memory wall* and handling *control dependencies*. In order to address these issues, *cache memories* and *branch predictors* are well-known hardware proposals that take advantage of, among other things, exploiting both *temporal memory reuse* and *branch correlation*. In other words, they try to exploit the *dynamic redundancy* existing in programs. This redundancy comes partly from the way that programmers write source code, but also from limitations in the compilation model of traditional compilers, which introduces unnecessary memory and conditional branch instructions. We believe that today's optimizing compilers should be very aggressive in optimizing programs, and then they should be expected to optimize a significant part of this redundancy away.

On the other hand, *optimizations performed at link-time* or directly applied to final program executables have received increased attention in recent years, due to limitations in the traditional compilation model. First, even though performing sophisticated interprocedural analyses and transformations, traditional compilers do not have the opportunity to optimize the program as a whole. A similar problem arises when applying profile-directed compilation techniques: large projects will be forced to re-build every source file to take advantage of profile information. By contrast, it would be more convenient to build the full application, instrument it to obtain profile data and then re-optimize the final binary without recompiling a single source file.

In this thesis we present new *profile-guided compiler optimizations* for eliminating the redundancy encountered on executable programs at binary level (i.e.: *binary redundancy*), even though these programs have been compiled with full optimizations using a "state-of-the-art" commercial compiler. In particular, our *Binary Redundancy Elimination (BRE)* techniques are targeted at eliminating both *redundant memory operations* and *redundant conditional branches*, which are the most important ones for addressing the performance issues that we mentioned above in today's microprocessors. These new proposals are mainly based on *Partial Redundancy Elimination (PRE)* techniques for eliminating partial redundancies in a path-sensitive fashion. Our results show that, by applying our optimizations, we are able to achieve a 14% execution time reduction in our benchmark suite.

In this work we also review the problem of *alias analysis* at the executable program level, identifying why memory disambiguation is one of the weak points of object code modification. We then propose several alias analyses to be applied in the context of link-time or executable code optimizers. First, we present a *must*-alias analysis to recognize memory dependencies in a *path-sensitive* fashion, which is used in our optimization for eliminating redundant memory operations. Next, we propose two *speculative may*-alias data-flow algorithms to recognize memory independencies. These may-alias analyses are based on introducing unsafe speculation at analysis time, which increases alias precision on important portions of code while keeping the analysis reasonably cost-efficient. Our results show that our analyses prove to be very useful for increasing memory disambiguation accuracy of binary code, which turns out into opportunities for applying optimizations.

All our algorithms, both for the analyses and the optimizations, have been implemented within a *binary optimizer*, which overcomes most of the existing limitations of traditional source-code compilers. Therefore, our work also points out the most relevant issues of applying our algorithms at the executable code level, since most of the high-level information available in traditional compilers is lost.

Acknowledgments/*Agradecimientos*

I want to thank all the people who provided guidance, help, and support while I was working on this thesis.

Most of all, I am indebted to my advisor Roger Espasa, for his continuous support, great patience, and specially for let me enough freedom to do things I wanted to do the way I thought they should be done. Without his guidance and encouragement this thesis would have never been possible.

I would also like to thank the members of my thesis committee for the effort they put into judging this thesis. Special mention to Professor Jordi Cortadella, for his suggestions and helpful discussions on some ideas that significantly improved this document.

I am specially indebted to Professors Saumya Debray and Cristina Cifuentes, for the chance to work with them, and for their kindness hosting me during my stays at the University of Arizona and Sun Microsystems, respectively. The numerous insightful discussions with them about different research topics have definitely contributed this thesis. My gratitude also to Brian Lewis, Sri Nair, Greg Wright, and everyone at Sun Microsystems Laboratories. And of course, to Emiliano Bartolomé, Malen Flaquer, and Pere Obrador, for making me feel at home during my visits to California.

I am also specially grateful to the people at the Department of Computer Architecture. Thanks to Professors Eduard Ayguadé, Antonio González, and Mateo Valero, for their encouragement in so many different situations, I would also thank to Agustin Fernández, Jordi García, Toni Juan, Josep Lluis Larriba, and Josep Llosa, for their help and friendship along these years. Special thanks to Xavier Vera, who carefully read a draft of this thesis and suggested many improvements. My gratitude also to the administration, LCAC, and CEPBA staff, for their excellent administration an technical support.

Finally, I would like to express my thanks to everyone I have not cited above but has help me, directly or not, in the long way until this thesis has been finished.

Todo proceso sufre una gran influencia del entorno donde se elabora. Por ello, las personas que me han rodeado en los últimos años, tanto a nivel profesional como a nivel personal, han influido en gran medida en la elaboración de este trabajo.

Quiero expresar mi agradecimiento a todos los que han sido mis compañeros y amigos en el Departamento de Arquitectura de Computadores, muy especialmente a Jaume Abella, Yolanda Becerra, Ramon Canal, Jesús Corbal, Julita Corbalán, Pepe González, Larisa Miranda, Daniel Ortega, Maite Ortega, Jesús Sanchez, Gladys Utrera y Xavi Vera. Con ellos he compartido no sólo un entorno de trabajo extraordinario, sino también incontables confidencias y vivencias que no podré olvidar nunca. Todo hubiera sido mucho más difícil sin ellos.

Este trabajo no podría haberse desarrollado sin una base sólida en el aspecto educativo, personal y humano. Por ello quiero agradecer a mi familia, especialmente a mis padres Domingo y Josefa, y a mis hermanos Miguel Angel y Merche, el apoyo y cariño que siempre me han brindado. Sin ellos nunca habría llegado tan lejos.

Por último, mi más profundo agradecimiento a Laura, más de lo que aquí podría expresar con palabras. Por su apoyo y ánimo incondicionales, y ante todo por darme la estabilidad necesaria para acabar este trabajo. En definitiva, por todo el amor, el cariño y la alegría que he recibido de ella durante estos años.

This work has been supported by the Intel Labs Barcelona (BSSAD), the Spanish Ministry of Science and Technology, and the European Union (FEDER funds), under grants PN98 46057403-1, CYCIT TIC98-0511 and TIC2001-0995-C02-01.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we outline the motivations behind this thesis. We first identify the memory wall *and the* control dependencies *as the major performance factors in today's microprocessors. Then, we define the concept of* binary redundancy*, that is, the dynamic redundancy observed in binary programs related to these performance issues. We will see that binary redundancy opens new opportunities for compiler optimizations, and how this redundancy removal appropriately fits within link-time or executable code optimizers. Finally, we give a brief description of our objectives and describe the structure of this work.*

1.1 Motivation

Superscalar processors represent the major trend in high-performance processors in the last several years [SS95]. These processors naturally evolve from pipelined architectures, and try to obtain higher performance rates by simultaneously fetching and executing in parallel several independent instructions in a single cycle. The two major performance limiters for today's superscalar processors are:

The *memory wall* Overcoming the long latencies caused by the different speed between the processor and the memory subsystem is one of the primary issues for high-performance computing [WM95]. To overcome this problem, the best known hardware solution relies on the use of *cache memories* [Smi82]. Memory is then organized hierarchically to take advantage of *program locality*[1], by exploiting both *spatial locality* (i.e., the nearby data to that being used is likely to be accessed in the future), and *temporal reuse* (i.e., the data being used is likely to be accessed again in the future).

Unfortunately, due to Moore's Law [Moo65], the memory wall is growing for each generation of new processors. That's why, even for memory accesses to the first-level cache, memory latency has grown beyond a single cycle in modern processors. As an example, 3 cycles are needed in order to access the 64Kb L1 cache in the Alpha 21264 processor [Kes99].

Control dependencies Pipelined processors introduce the problem of fetching one instruction per cycle without waiting for the previous instructions to finish [HP96]. When such instruction is a branch, its outcome is unknown until one or more cycles after the instruction was fetched, making the program control flow uncertain at that point and introducing execution bubbles in the pipeline. This situation is even a major performance issue in superscalar processors due to longer pipelines [SS95].

For handling control dependencies, the best known hardware proposals are based on *predicting* the behavior of conditional branch instructions before the branch is actually executed [Smi81, YP92, CG94, LCM97]. This behavior is defined by the *branch target address*, and the *branch direction*, in case of a conditional branch. Then, the next path is speculatively fetched the next cycle with no delay. In case of misprediction, it will be necessary to squash the wrongly fetched instructions, possibly incurring in additional penalties.

Even though hardware approaches have shown to be very valuable in current superscalar processors, they are not enough to overcome the performance factors pointed out before. From

[1]We only address in this work the discussion of *data caches*, since we only want to reason about program data. *Instructions caches*, for which several solutions have been proposed both in hardware [CMMP95, RBS96, FPP97] and software [PH90, GBSC97, RLPN$^+$99], are beyond the scope of this work.

our point of view, the compiler can and should help in *overcoming* these issues. The reason is that, as we will show in this work (see Chapter 5 and Chapter 6), it seems that there is still room for improvement in compiled code. Furthermore, the compiler approach is attractive for two reasons:

1. It has a null hardware cost, since it does not require additional transistors.
2. It provides performance improvements on already existing architectures.

The disadvantage is that the compiler is blind to run-time information. However, we believe that today's optimizing compilers should be very aggressive in applying sophisticated optimizations [Muc97a], in order to minimize the remaining negative impact of the above issues in program performance.

1.1.1 Binary redundancy

As we mentioned earlier, cache memories and branch predictors try to bridge the performance gap due to the negative impact of the memory wall and the control dependencies in current superscalar processors. These hardware structures do take advantage of several factors: among other things, cache memories exploit *temporal memory reuse*, while branch predictors exploit *branch correlation*. In other words, they try to exploit the *dynamic redundancy* existing in programs. In this work, we will be specifically concerned with two types of redundancy:

Memory redundancy This refers to memory instructions that access memory cells already referenced in the past, and that, obviously, increase the temporal reuse of the program.

Conditional branch redundancy That is, conditional branches whose outcome can be statically determined along some paths of the program at compile time.

It is straightforward to note that, unlike more general computation redundancy [MR79, KRS94a, Muc97b, BGS98], these redundancy types are closely related to how binary programs are executed on superscalar processors, since there are hardware structures that try to deal with them explicitly (again, cache memories and branch predictors are good examples). For this reason, we define *binary redundancy* as the redundancy encountered on executable programs at binary level, which refers to both memory and conditional branch redundancy.

In general terms, binary redundancy exposes unnecessary recomputation at program run time of values that are already known, because they have already been computed in the program (by some other instruction or by a prior execution of the same instruction), or because they can be computed at compile time. As a result, binary redundancy exposes optimization opportunities that can be potentially exploited by an optimizing compiler in order to remove these redundancies away.

(a)
```
if (*p > 0)
{
    *q = ...
}
if (*p == 1)
{
    ...
}
```

(b)
```
register r = *p;
if (r > 0)
{
    *q = ...
}
if (r == 1)
{
    ...
}
```

(c)
```
register r = *p;
if (r > 0)
{
    *q = ...
    if (r == 1)
    {
        ...
    }
}
```

Figure 1.1: Example of memory and conditional branch redundancies, and its elimination: (a) original code, (b) resulting code after eliminating the redundant memory reference, and (c) resulting code after eliminating the redundant conditional branch.

Figure 1.1 illustrates a C code example where cases of memory and conditional branch redundancy can be observed:

- Looking at Figure 1.1a, we can see that memory location referenced by `*p` is redundantly accessed in the two conditionals. To remove the redundant memory reference (i.e., the second `*p`), the key idea is to reuse the first loaded value rather than load the same value again, by keeping the value in a register while it is needed (as it can be seen in Figure 1.1b).

- On the other hand, condition `*p == 1` is known to be false when condition `*p > 0` evaluates to false. In this example, it means that the outcome of the second conditional branch can be determined sometimes, depending on the outcome of the first conditional branch. To remove such conditional branch redundancy we need in this case to restructure the program, in order to isolate the correlated path form the non-correlated one (as shown in Figure 1.1c).

It is important to note that for removing both memory and conditional branch redundancies, neither pointer `p` must change between the two `*p` read accesses, nor pointer `q` must be aliased with pointer `p` in the example. Therefore, in order to detect and eliminate binary redundancies, the need for disambiguating memory references will make *alias analysis* a critical factor [ASU86a, Muc97c, GLS01].

Binary redundancy comes partly from the way that programmers write source code, but also from limitations in the compilation model of traditional compilers [ASU86b]. Thus, for example, a variable may not have been kept in a register by the compiler because it was a global, or because the compiler was unable to resolve aliasing adequately, or because there were not enough free registers available. Similarly, the compiler may not have the opportunity to perform intraprocedural analyses and optimizations, thus continuously evaluating conditional branches that could be statically determined otherwise. These constraints are often responsible for a significant number of unnecessary memory and conditional branch instructions in binary programs [Bod99].

In summary, binary redundancy exists even in reasonably well-written programs, but is often hard to remove. In general, the removal of redundant instructions may speed up the program in two ways:

1. When the redundant instruction is removed, there are fewer instructions to execute. As a result, there is less contention for hardware resources (e.g., functional units, registers, cache ports, etc.), which allows scheduling the remaining instructions earlier. Note that the resource constraints restriction could be overcome with wider processors, making this benefit of somewhat less important for future high-performance processors.

2. Because the reused value is available sooner than the recomputed one, subsequent instructions that need the value can be scheduled earlier. Essentially, removing an instruction breaks some paths of data dependences among instructions. When the redundancy removal breaks the critical path of a program, it may be possible to schedule such program in fewer machine cycles. Observe in this case that the critical path constraint is a manifestation of the data-flow limit of the program [Wal91], and hence cannot be overcome without program transformation, making this restriction more important for future processors.

Note that the instruction schedule is improved regardless of whether it is created statically (i.e., by a compiler) or dynamically (i.e., by an out-of-order processor). Binary redundancy elimination is thus beneficial for both statically and dynamically scheduled processors.

1.1.2 Binary optimizations

Optimizations performed at link time or directly applied to final program executables have received increased attention in recent years [SW92, RVL+97, Goo97, CGLR97, MDWdB01, SDAL01, SDA02, LMP+04], mainly due to limitations in the traditional compilation model. Large programs tend to be compiled using separate compilation, that is, one or a few files at a time. Therefore, the compiler does not have the opportunity to optimize the program as a whole, even when performing sophisticated interprocedural analyses and transformations. Furthermore, basic knowledge about the program objects (e.g., whether a variable is stored on the heap, stack or global area) is also lost when moving from one file or compilation unit to the next.

Vendors have tried to overcome these limitations by compiling separate files that contain intermediate representations rather than final object code [Wal86, Sil99]. Later, at link time, these "fake" intermediate object files are fully compiled and optimized together with the rest of the program units. The drawback of this approach is that it does not mix well with traditional Makefile-based software development environments. Furthermore, the implemented analyses and optimizations are possibly limited to only code that is available for examination at compile time. This means that code involving calls to procedures defined in separately compiled modules, and to dynamically dispatched "virtual functions" in object oriented languages (in the case where the virtual function is never overridden) cannot be effectively optimized. This is even valid for interprocedural analyses and optimizations performed by sophisticated optimizing compilers [JNMW00, LCH+03], since the source code of programs is not always available such as in old legacy software or library code. As a consequence, link-time or executable code optimizers that are based solely on the final object representation (i.e., when the *entire* program is available for inspection) have the attraction of being able to work on a full program basis and be fully integrated on a normal compile-build-test cycle.

A second reason for the recent interest in binary optimization has been the emergence of profile-directed compilation techniques [PH90, CMCH92, CL96, GBF98, FLM+01, GMZ02]. As it has been shown in several studies [Sar89, BL96, CFE97, BMS98], the compiler can use to great advantage the profiling information to identify new optimization opportunities that are frequently observed during program execution but are not detected by a simple static analysis. However, the same problem of separate compilation plagues the production use of profile feedback. After a previous run of the program to collect profile data, large projects will be forced to re-build every file to take advantage of profile information. Furthermore, the profile-collection phase needs to be specially coded into the Makefile environment. By contrast, it would be more convenient to be able to build the full application, instrument it to obtain profile data and then re-optimize the final binary without recompiling a single source file. This is the approach taken by Spike [CGLR97, FLM+01], for example, and is only possible if using binary optimization techniques.

Binary optimizers overcome most of the existing limitations of traditional source-code compilers. However, since most of the high-level information available in traditional compilers is lost, working at executable code level has its own set of issues:

- Machine code usually has much less semantic information than source code, which makes it much more difficult to discover control-flow or data-flow information.

- Traditional compiler analyses are usually carried out on representation of source programs in terms of high-level language constructs that often ignore "nasty" features typically encountered in executable programs (e.g., type casts, obfuscated pointer arithmetic, out-of-bounds array accesses, etc.), since such features result in non-standard conforming programs whose behavior are not guaranteed to be preserved under compiler optimizations.

- Executable programs tend to be significantly larger than the original source programs they were derived from.

In most cases, sophisticated analyses that are practical at the source level (which usually operate on a per module or per function basis) turn out to be of limited utility at the executable code because of their time or space requirements. When working on executable code, therefore, it is often necessary to make tradeoffs between precision and efficiency.

1.2 Thesis overview

In this section we provide a brief description of the topics we deal with in this thesis dissertation. We present the problems we are trying to solve, the approaches we take to solve them, and the structure of this work.

1.2.1 Thesis objectives

In this thesis we present new *profile-guided compiler optimizations* for eliminating the *binary redundancy* encountered on executable programs at binary level, even though these programs have been compiled with full optimizations using a "state-of-the-art" commercial compiler. Our *Binary Redundancy Elimination (BRE)* optimizations will be targeted at eliminating both *redundant memory operations* and *redundant conditional branches*, which are the most important ones for addressing the major performance issues in today's microprocessors. The main contributions of this work are the following:

Computing alias information We first review the problem of *alias analysis* at the executable program level, identifying why memory disambiguation is one of the weak points of object code modification. Then, we present several alias analyses to be applied in the context of link-time or executable code optimizers. First, we propose a *must-alias* analysis to recognize memory dependencies in a *path-sensitive* fashion (this scheme will be used in our optimization for eliminating redundant memory operations). Next, we present two *speculative may-alias* data-flow algorithms to recognize memory independencies. These may-alias analyses are based on introducing unsafe speculation at analysis time, which increases alias precision on important portions of code and keeps the analysis reasonably cost-efficient. Our results will show that our analyses prove to be very useful for increasing memory disambiguation accuracy of binary code, which translates into additional opportunities for applying optimizations.

Eliminating memory redundancies We discuss the discovery and elimination of memory operations that are redundant and can be safely removed in order to speed up a program, an optimization that we call *Memory Redundancy Elimination (MRE)*. We quantify these effects and show that a high percentage of memory references at program run time can be considered redundant because they are accessing memory locations that have been referenced in a near past. We then present several profile-based MRE algorithms targeted at optimizing away these redundancies. These optimization algorithms are mainly based on *Partial Redundancy Elimination (PRE)* techniques for eliminating partial redundancies in a path-sensitive fashion. Our results will show that a significant amount of memory redundancy can indeed be eliminated, which translates into important reductions in execution time.

Eliminating conditional branch redundancies Finally, we propose several optimization techniques for detecting and eliminating redundant conditional branches on executable code. These are branches whose outcome can be determined at compile time, and thus they can be safely removed in order to speed up a program. We call this optimization *Conditional Branch Redundancy Elimination (CBRE)*. We first show that important amounts of conditional branches in a program can be considered redundant because their outcomes can be determined from a previous short dynamic execution frame. We also see how important memory disambiguation is in order to catch branch redundancy. Then, we present several CBRE algorithms targeted at optimizing away these redundancies. Our results will show that a significant amount of redundant conditional branches can be removed with moderate levels of code growth.

All our algorithms, both for the analyses and the optimizations, will be implemented within a *binary optimizer*, which overcomes most of the existing limitations of traditional source-code compilers. Therefore, our work also points out the most relevant issues of applying our algorithms at the executable code level, since most of the high-level information available in traditional compilers is lost.

1.2.2 Structure of this document

The work we present in this document is organized as follows:

- Chapter 2 introduces some terminology for program analysis and optimization we use in this work, and also presents a brief description of the most significant related work in this research field.

- Chapter 3 presents our research environment. We give an overview of our experimental framework at all levels (i.e., compilation, execution and simulation), and also introduce the benchmark suite and the methodology used for evaluation.

- Chapter 4 reviews the problem of alias analysis at the executable program level. We then present several alias algorithms to be applied in the context of link-time or executable code optimizers, that prove to be very useful for increasing memory disambiguation accuracy of binary code.
- Chapter 5 presents our techniques addressed to eliminate redundant memory operations on executable code. These proposals are mainly based on new applications of PRE techniques for eliminating partial redundancies in a path-sensitive fashion.
- Chapter 6 shows our optimizations targeted at removing conditional branch redundancies at binary programs. Our algorithms are even able to detect and eliminate partial conditional branch redundancies, applying code restructuring when needed.
- Finally, Chapter 7 summarizes the main contributions of this thesis and presents the open areas for future research.

Chapter 2

Background and related work

In this chapter we introduce some preliminary background on compiler analyses and optimizations in general, focusing on techniques related to the work presented in this document. We also review the state of the art on several topics: (a) profile-guided optimizations, (b) optimizations targeted at eliminating redundancy, and (c) binary optimizers. In particular, we discuss previous work closely related to what is developed in this thesis.

2.1 Preliminaries

We start this chapter by reviewing some basic compiler concepts related to the work presented in this dissertation. The rest of this work is developed from first principles [ASU86c, Muc97d].

2.1.1 Control flow graph

An important part of a compiler's internal representation relies on characterizing the control structure of a program, by transforming it into some intermediate form more suitable for further modifications. We define the control flow in a program as a rooted, directed graph with a set of nodes and a set of edges connecting such nodes.

Definition 2.1.1 *A* directed graph *G is a pair (N, E) where N is the node set and E is the edge set such that $E \subseteq N \times N$.* Immediate predecessor *and* successor *of nodes are defined with maps* pred *and* succ*, such that $pred(n) = \{m|(m, n) \in E\}$ and $succ(n) = \{m|(n, m) \in E\}$. A* finite path *of G is a sequence $\omega = n_1, \ldots, n_k$ of nodes such that $(n_i, n_{i+1}) \in E$ for all $1 \leq i < k$.* □

We then use a *Control Flow Graph (CFG)* as the underlying program representation because it is the most commonly used intermediate program representation in both production and research compilers.

Definition 2.1.2 *A* control-flow graph *$G = (N, E, start, end)$ is a directed graph (N, E), in which nodes $n \in N$ represent* basic blocks *containing individual program statements (instructions).* Edges *$(m, n) \in E$ represent the non-deterministic branching structure of the program. Nodes* start *and* end *are the unique start node and end node of G; they are assumed to have no predecessors and no successors, respectively. It is also assumed that every node $n \in N$ lies on a path $\omega = start, \ldots, end$. We also say that node m* dominates *n (written m dom n) if every possible path w from* start *to n includes m. Similarly, node n* postdominates *m (written n pdom m) if every possible path w from m to* end *includes n.* □

The above definition is valid for representing the flow graph of both a program and a *procedure* or *routine*. Because CFG directly exposes program's control-flow paths, it enables an intuitive and efficient formulation of code analysis and optimization, and thus become the standard representation in optimizing compilers.

2.1.2 Basic blocks

As we have mentioned, nodes in a CFG are called basic blocks. A *basic block* is, informally, a straight-line sequence of instructions that can be entered only at the first of them and exited only from the last of them. The first instruction in a basic block is called a *leader*.

A natural extension of a basic block is known as *Extended Basic Block (EBB)*, which is a set of basic blocks with either (a) a single entry point but multiple exit points, or (b) a single exit point but multiple entry points. The latter is also known as *reverse extended basic block*. An EBB can be thought of as a tree of basic blocks, where the EBB *root* corresponds to the single entry or exit point, respectively. Extended basic blocks are interesting in some contexts, where analyses and transformations are more effective when done on EBBs than on simple basic blocks.

2.1.3 Instruction set representation

For the machine code examples of this document, we assume for simplicity a canonical RISC instruction set. We can distinguish between the following type of instructions:

- Memory is accessed only through explicit load and store instructions, which have the form `load` $k(r_b), r_a$ and `store` $r_a, k(r_b)$. Memory instructions have the effect of reading/writing from/to the location whose address is $k + contents(r_b)$, where k is a constant offset and r_b is the base register.

  Two special registers, denoted as *sp* or *stack pointer*, and *gp* or *global pointer*, point to the program stack and global data areas, respectively.

- For arithmetic and logic instructions we assume the form $op\ src_1, src_2, dst$, where op denotes an operation, dst is a destination register, and src_1 and src_2 are source registers[1]. For instance, an `add` instruction computes the sum of src_1 and src_2 into dst. Many other operations can be expressed in terms of this form.

- To determine control-flow information, we consider the usual set of conditional branches, and direct and indirect unconditional branch instructions. For conditional branches, we have the "test and branch" form `b`$op\ src_1, src_2$, which means that the branch will be taken if the relationship between src_1 and src_2 holds the condition defined by the instruction opcode.

We assume this generic machine code instruction set to be the compiler intermediate language as well. Although this might seem to be very non-portable, this representation is in fact very similar to the low-level intermediate representation used in many compilers [ASU86d, Mah92, Muc97e, Mor98].

[1]To simplify the discussion we abuse notation and allow either src_1 or src_2 to be an integer constant, denoting an immediate operand. We also allow the lack of one of the source operands so that register moves can be modeled.

2.2 Data-flow analysis

Before actually applying code transformations, compilers perform a variety of analysis to find out how program data is being used [BGS94]. The purpose of *data-flow analysis* is to provide such information, and serve as a a bridge between the program representation and the actual program optimization. This section describes the most important ideas of applying data-flow analysis for understanding the behavior of a program.

Performing data-flow analysis Data-flow analysis is usually performed on a control-flow graph. It attempts to track the flow of data through the program's variables and to characterize the values of variables at various points of execution. By traversing each path, the analysis verifies the algebraic rules posed by the representation (e.g., whether a variable has been redefined). Thus, by being a tool for summarizing global program properties, data-flow analysis identifies patterns, which are then used to guide the program transformation phase of a particular optimization.

There are two primary strategies for performing data-flow analysis:

1. Develop a set of equations that are applied to basic blocks, which yields a list of variables that match some desired criteria [Muc97f]. For example, suppose we wish to compute the set of values that are available at the exit of a given basic block B. Conventionally, this set is name *out*. The equation to compute $out(B)$ is

$$out(B) = gen(B) \cup (in(B) - kill(B)) \tag{2.1}$$

   where $gen(B)$ is the set of values generated inside the block B, $in(B)$ is the set of values that *reach* B from some other block, and $kill(B)$ is the set of values that were overwritten in B (by an assignment statement, for example). A straightforward method for computing the data-flow information, which is also the most widely used, is to apply Equation 2.1 iteratively until reaching a fixed point [ASU86a].

2. Perform data-flow analysis based on *chains* [Muc97g]. The idea is to identify where variables are assigned a value (called a *definition*) and where the value is used (a *use*). By connecting definitions with uses the compiler can perform a wide variety of analysis. Thus, either uses will be connected to definitions or definitions will be connected to uses (and even both in some cases).

In both cases, we must be certain that a data-flow analysis computes information that does not misrepresent what the program being analyzed does, in the sense that it must not indicate that a transformation of the code is safe to perform that, in fact, is not safe. We must guarantee this by careful design of the analysis and by being sure that the solution computed is, if not an exact representation of the program's manipulation of its data, at least a conservative approximation of it. However, to obtain the maximum possible benefit from optimization, we seek to pose data-flow analyses as aggressive as we can make them. Thus, we shall always attempt to walk the fine line between being as aggressive as possible in the information we compute and being conservative, so as to get the greatest possible benefit form the analyses and code transformations we perform without ever modifying program's semantic.

Characterizing data-flow analysis Data-flow analysis can be categorized along several dimensions, including the following [NNH99]:

Flow direction Corresponds to the direction of information flow. Almost all the data-flow problems are one-directional, either *forward* (i.e., in the direction of program execution) or *backward* (i.e., opposite to the direction of program execution). *Bidirectional* problems require forward and backward propagation at the same time and are significantly more complicated to formulate, understand, and solve than one-directional problems. Fortunately, bidirectional data-flow problems are rare.

Flow sensitivity The *flow-insensitive* versus *flow-sensitive* classification indicates whether a data-flow analysis is independent of the control flow encountered or not. This distinction is important because it determines the computational complexity of the problem under consideration. Flow-insensitive problems can be solved by solving subproblems and then combining their solutions to provide a solution for the whole problem, independent of control flow. Flow-sensitive problems, on the other hand, require the algorithm to follow the control-flow paths through the flow graph to compute the solution.

May vs. must information It is useful to distinguish *may* information from *must* information. The former indicates what may occur on *some path* through a flow graph, while the latter indicates what must occur on *all paths* through the flow graph. The may vs. must classification is important because it indicates whether a property must hold, and hence can be counted on, or that it only may hold, and so must be allowed for but cannot be counted on.

Intra- vs. inter-procedurality Compilers often limit their program analyses to individual procedures. That is, they are *intraprocedural*, which means they are applied without regard to the calling context in which that procedure is used or the procedures it calls. As a result, one must generally assume that a called procedure may use or change any variable it might be able to access, which clearly inhibits optimization opportunities. By contrast, *interprocedural* approaches are ones that use the calling relationships among a set of procedures to drive the analysis. The intra- versus inter-procedural distinction can also be applied when reasoning about compiler optimizations.

Context-sensitivity Interprocedural data-flow analyses can be either *context insensitive* or *context sensitive*. Context-insensitive analyses simply combine the control-flow graphs for individual procedures into a large graph and analyze this using standard intraprocedural techniques, without keeping track of which return edges correspond to which call edges. This has the advantages of simplicity and efficiency: nothing special needs to be done to handle interprocedural control flow, and a procedure does not have to be re-analyzed for its various call-sites [CR82]. The problem is that such analyses can suffer from a loss of precision because they can explore execution paths containing call/return pairs that do not correspond to each other and therefore cannot occur in any execution of the program. Context-sensitive analyses, by contrast, avoid this problem by maintaining information about which return edges correspond to which call sites, and propagating information only along realizable call/return paths [EGH94]. The price paid for this improvement in precision is an increase in the cost of the analysis.

We next discuss the most popular data-flow analysis strategies for understanding the behavior of a program. We also give additional information on the corresponding existing techniques for analyzing executable code.

2.2.1 Liveness analysis

Liveness analysis is a well-known technique employed by most compilers to guide optimizations based on transforming variables [ASU86a, Muc97f], such as useless code elimination (see Section 2.3.1.2) and register allocation [Muc97h]. The analysis attempts to determine whether a value kept in a variable or storage location may be used later on during program execution. A variable is said to be *live* at a particular point in a program if there is a path to the exit of a given block of code along which its value may be used before it is redefined. It is *dead* if there is no such path. A simple backward data-flow analysis is usually performed to compute this information at each point in a flow graph.

Liveness analysis can also be performed on executable code if we let registers take the place of variables [SW92, Goo97]. The problem is in this case simplified by the fact that, commonly, there is no aliasing between registers, and the number of registers for a given processor is bounded by a constant. As a result, *register liveness analysis*, unlike traditional liveness analysis which is usually intraprocedural, can be now applied interprocedurally or in a context-sensitive fashion in order to achieve higher precision rates with a moderate increase in the cost of analysis [Mut98]. Nevertheless, what makes the analysis difficult are anomalies of the control-flow graph and scalability issues, which are the common problems that appear when working at executable code level (as already mentioned in Section 1.1.2).

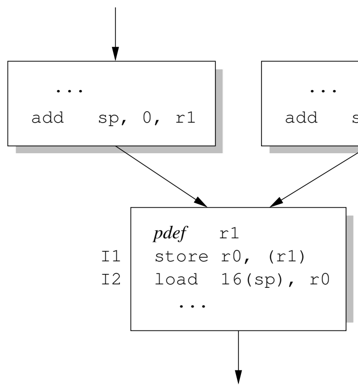


Figure 2.1: Sample code where different definitions are reaching a use. A *pseudo definition* is introduced for register r_1.

2.2.2 *Use-def* chains

Use-def chains are a sparse representation of data-flow information about variables, which simplifies the implementation of several well-known optimizations, such as common subexpression elimination [ASU86a]. A *ud-chain* for a variable connects a use of that variable to all the definitions that may flow to it. Abstractly, a ud-chain is a function from a variable and a basic-block-position pair to sets of basic-block-position pairs, where every element corresponds to each definition [Muc97f].

As far as executable code is concerned, *register use-def chains* is an analysis that provides, for each use of a machine register, a pointer to its definition. The ud-chains form a directed graph whose nodes are instructions and whose edges are use-def pointers. When there are several definitions of a register reaching a use, as depicted in Figure 2.1, it is common to introduce a *pseudo instruction* at an appropriate confluence point which also defines that register, thereby shadowing the other definitions. This is analogous to ϕ functions used with the *Static Single Assignment (SSA)* form [CFR$^+$91], although pseudo insertions in this case result in "less accurate" use-def information. All registers are enforced to be defined before they are used by inserting pseudo instructions at all entry nodes of the graph.

2.2.3 Alias analysis

Alias analysis refers to the determination of all the storage locations that may be accessed in two or more ways [ASU86a]. For example, a variable in a C program may have its address computed and be assigned to or read from both by variable name and through a pointer. Determining the range of possible aliases in program is essential for optimizing it correctly, while minimizing the sets of aliases found is important for doing optimizations as aggressively as possible.

At the machine code level, the problem of alias analysis or *memory disambiguation* is to statically determine the relationship of every pair of memory references in a program. A *reference* typically identifies a memory address and an access size. Then, for two particular references, there are four possible answers that an alias disambiguator can return:

- They are *identical*, which means that both references always point to the same location.
- They are *intersecting*. This means that memory accessed by both references partially overlaps.
- They are *disjoint*, which means that they are never aliased, and therefore, independent.
- *Unknown*. That is, the disambiguator cannot determine statically the relationship between the two references.

The aliasing problem can be formulated by a combination of *may-alias* analysis, which answers whether two memory references are independent, and *must-alias* analysis, which checks references for memory dependencies. Performing *no alias analysis* conservatively leads to the assumption that every load and store instruction is always dependent on every previous store instruction.

While there is an extensive body of work on pointer alias analysis of various kinds [WL95, SH97, DWM98, CH00, GLS01], these are mostly high-level analyses carried out in terms of source language constructs that turn out to be of limited utility at the machine code level. In fact, the problem of memory disambiguation is one of the weak points of object code modification, because important information typically available in an ordinary compiler is lost at executable code level, where the contents of every register is potentially an address.

2.2.3.1 Alias analysis by instruction inspection

For disambiguating references, a common technique in compile-time instruction schedulers is *alias analysis by instruction inspection* [Muc97g]. Here, two memory references are considered within an extended basic block to see if it is obvious that they point to either the same or different memory addresses. For example, independence between instructions I_1 and I_2 in Figure 2.2 can be proved if either of the following conditions hold:

- Different memory regions are referenced. For example, one of the instructions uses a register known to point to the stack and the other uses a register known to point to the global data area, as shown in Figure 2.2a.
- They access data at addresses $k_1(r_1)$ and $k_2(r_2)$, as shown in Figure 2.2b. Base registers r_1 and r_2 are computed by two (possibly empty) sequences of instructions such that

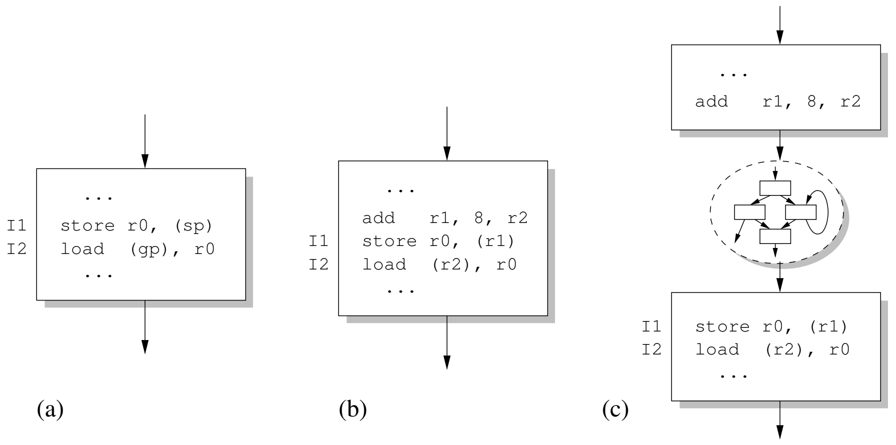


Figure 2.2: Sample code with different techniques for may-alias disambiguation by instruction inspection: (a) knowing that accesses point to different memory regions; (b) using register copies and address arithmetic propagation; (c) general case beyond extended basic block boundaries, using *use-def* chains.

$r_1 = c_1 + contents(r_0)$ and $r_2 = c_2 + contents(r_0)$, for some register r_0. Both accesses are non-aliased if both chains use the same definition of r_0, and $c_1 + k_1$ and $c_2 + k_2$ do not overlap[2]. To detect the definition of register r_0 a simple backwards data-flow algorithm may be used.

All other memory instruction pairs are considered to be aliasing. Unfortunately, this simple approach does not work if information about register copies and address arithmetic needs to be propagated across extended basic block boundaries. To do so, *register use-def chains* are required, as presented in Section 2.2.2. In the general case, an instruction inspection algorithm tries to derive a symbolic description for each memory instruction and then compare these descriptions for checking independency, as shown in Figure 2.2c.

2.2.3.2 Residue-based global alias analysis

Instruction inspection fails when several definitions are reaching a use. For example, looking back to Figure 2.1, register r_1 is defined with two possible stack values. However, possible locations accessed at instruction I_1 are disjoint with respect to the location accessed by instruction I_2. Debray *et al.* [DMW98] propose an interprocedural algorithm to reason about may-alias information of executable code. The analysis, which is implemented in the context of a link-time optimizer, can handle complex pointer arithmetic and features usually ignored by traditional alias analysis algorithms.

[2]In case that $c_1 + k_1 = c_2 + k_2$ the two references point to the same location, which indicates a dependency between the two accesses.

An alias analysis will in general associate each register with a set of possible addresses at each program point. The basic idea of the algorithm is to reason about arithmetic computations modulo some pre-selected value k. A set of addresses is then represented by an *address descriptor*, which is a pair $\langle I, S\rangle$, where I is the *defining instruction* for a machine register r, and S is a set of mod-k residues with respect to the value computed by instruction I. This representation can then distinguish between addresses involving distinct "small" displacements (i.e., less than k) from a base register. Comparing descriptors can be reduced to a comparison of mod-k sets, using dominator information to handle loops correctly. Since k is fixed, S can be represented as a bit vector of length k. Their implementation corresponds to mod-k residues with $k = 64$, in part determined by the fact that the set of mod-k residues for this choice of k corresponds to a bit vector that fits exactly in one 64-bit machine word. This means that set operations such as union, intersection, checking containment, etc., are compactly representable and can be carried out in $O(1)$, which is cheap enough to be practical for the analysis of large binaries.

As far as the analysis is concerned, a data-flow system is used to propagate values through the control-flow graph. They use a conservative operation to "merge" the information coming along the incoming edges at vertices in the interprocedural control-flow graph [CC77]. Thus, if the values for a register r being propagated along two incoming edges at a vertex in the flow graph are described by address descriptors $\langle I_1, S_1\rangle$ and $\langle I_2, S_2\rangle$ respectively, and $I_1 \neq I_2$, then the information about r is generalized to the conservative value $\perp$ [NNH99], denoting a total lack of information. The essential idea behind this operation is to associate a single descriptor with a register at each program point of interest, rather than a set of descriptors, keeping the memory requirements of the analysis reasonable: for each basic block one address descriptor per register is needed, corresponding to the *out* register set at the exit of the basic block[3]. For a given choice of value k, the analysis requires $RN(k + w)$ bits of memory for a program with N basic blocks on a machine with R registers, where w is the number of bits per machine word.

Amme *et al.* [ABZT98] present a method to detect data dependencies in assembly code by using symbolic value propagation. However, the algorithm does not work beyond procedure boundaries, and symbolic values are not propagated through memory when registers are saved and restored. Although it has been applied to assembly code, it is not obvious that using the algorithm for interprocedural whole-program analysis would scale up to problems of this size.

2.2.4 Abstract interpretation

Abstract interpretation [CC77, CC79] is a mathematical approach to statically analyze the dynamic properties of software applications at compile time, without executing the program

[3]The *in* register set can be computed easily from the *out* register sets of its predecessors.

itself. It may be seen as an extension of compilation techniques that enable compilers to predict how a program will behave, before actually executing the application. Debray [Deb95] discusses the role of abstract interpretation in low-level compiler optimizations.

In the spirit of abstract interpretation, several methods have been proposed for obtaining dynamic program properties by using *symbolic evaluation* and simple algebraic rules [Rau91, TP95, ABD+02, DLS02]. Some of these techniques have been focused on a wide variety of program optimizations based on propagating information about value ranges of variables [Pat95, SBA00, BGSW00, MRS+01, CGS04].

2.3 Compiler optimizations

After analyzing the code, the compiler can begin to transform it. The importance of compile-time code optimizations to improve code efficiency has been recognized for many years [ASU86a, Muc97i]. Some transformations may enable others which in turn enable the original transformation to improve the code further. The compiler designer must decide on an order in which to apply code optimizations.

The primary objective of a traditional optimizing compiler is to reduce the number and complexity of the instructions executed by the processor. Superscalar processors can potentially achieve large performance improvements by exploiting *Instruction Level Parallelism (ILP)*, which refers to the ability of executing *in parallel* low-level machine instructions (e.g., memory loads and stores, integer adds, floating point multiplies, etc.). As a result, the amount of ILP available to superscalar processors can be limited with conventional compiler optimization techniques, which are initially designed for scalar processors. As the amount of parallel hardware within processors continues to grow, optimizing compilers will be required to also expose increasing levels of ILP to effectively utilize the parallel hardware [MCG+92, BGS94].

In general, in doing optimizations, compilers attempt to be as aggressive as possible in improving program code, but never at the expense of making it incorrect. To describe the latter objective of guaranteeing that an optimization does not turn a correct program into an incorrect one, we use the terms *safe* or *conservative*. Thus, when a program is optimized, the optimizations must be conservative enough so that the semantics of the program remains unchanged [BGS94].

2.3.1 Base optimizations

We next review a list of common optimizations which are widely used in optimizing compilers. We also give additional information on the corresponding existing techniques for optimizing executable code.

2.3.1.1 Optimization of constant expressions

Optimizing constant expressions is one of the most important optimizations that a compiler can perform, and optimizing compilers will do so aggressively [CCKT86, WZ91, CH95]. *Constant propagation* is a transformation that, given the assignment of a constant value c to a variable v, replaces later uses of v with uses of c as long as intervening assignments have not changed the contents of v. *Constant folding* is a companion to constant propagation; when an expression contains a computation on values known to be constants, that computation is performed at compile time.

Programs typically contain many constants. By propagating and folding them simultaneously through the program, the compiler can do a significant amount of precomputation. More importantly, the propagation of constant values reveals many opportunities for performing other optimizations. In addition to obvious possibilities like dead code elimination (see Section 2.3.1.2) or reducing register pressure, loop optimizations are much affected because constants often appear in their induction ranges.

There are generally more opportunities for interprocedural constant propagation at link time than at compile time [Mut99]. The reason is that the entire program, including all the library routines, is available for inspection. Constants can be then propagated across compilation unit boundaries and even source language boundaries. Furthermore, at link time it is possible to detect and deal with architecture-specific computations that are not visible at the intermediate code representation level typically used by compilers for most optimizations. Muth *et al.* [MDWdB01] present a flow-sensitive interprocedural constant propagation algorithm at link time that is limited to register contents, which is able to deal with recursion, interpretation of conditionals, and propagation of constants through read-only data sections. De Sutter *et al.* [dSdBdBD01] improve this algorithm in a context-sensitive fashion. Both are able to deal with *strength reduction* (i.e., find a cheaper instruction to perform the same computation) in order to compute values into registers that are known to be constants [SW92].

2.3.1.2 Dead/unreachable code elimination

Dead code elimination removes instructions that can be proven to have no effect on the result of a computation [ASU86a]. That is, an instruction is *dead* if it computes a value that is not used (i.e., it is not *live*) on any executable path leading from the instruction. This is an important optimization, not only because some programs contain dead code as originally written, but also because many of the other optimizations create dead code. For this reason, it is usually applied several times during an optimizing compilation. As far as optimizing executable code is concerned, implementation of dead code elimination is often solely based on register liveness information.

Dead code elimination is often applied coupled to *unreachable code elimination*. A code fragment is *unreachable* if there is no control-flow path to it from the rest of the program. Code that is unreachable will never be executed, and can therefore be eliminated without affecting the behavior of the program. Unreachable code typically arises at compile time due to user constructs (e.g., debugging statements that are turned off by setting a flag) or as a result of other optimizations. On executable code, it arises primarily from the propagation of information across procedure boundaries. Implementation involves a straightforward depth-first traversal of the control-flow graph, and can be performed as soon as the CFG of the program has been computed.

2.3.1.3 Copy propagation and register renaming

Other optimizations may cause the same value to be copied several times. *Copy propagation* propagates the original name of the value and eliminates redundant copies [ASU86a]. It can be applied on intermediate code at any level from high to low. On the other hand, *register renaming* is a similar transformation but opposite in nature: it tries to assign unique registers to different definitions of the same register [Muc97j]. Renaming is in general applied to low-level intermediate code, before code scheduling.

On executable code, both transformations are limited to registers, although they can be improved by exploiting the use of the small displacement value in place of the second operand register allowed in most of the RISC instructions [Mut99]. Copy propagation reduces register pressure and eliminates redundant register-to-register move instructions. Thus, it tries to reduce the number of instructions in a program. The aim of register renaming is remove unnecessary false register dependencies so that flexibility available to code scheduling can be increased.

2.3.1.4 Procedure inlining and cloning

Procedure inlining is an important optimization that replaces a procedure call with a copy of the body of the called procedure, replacing each occurrence of a formal parameter with its corresponding actual parameter [Sch77, AJ88]. Inlining removes the function call/return costs and the overhead of argument passing. Besides, intraprocedural optimizations blocked by the procedure call boundary can be applied straightforwardly to the combined code of caller and callee, therefore improving their results. Another common technique for exploiting interprocedural information is *cloning*: the duplication of a callee so that its body may be specialized for the circumstances existing at a particular call site or set of call sites [CHK93].

Despite its benefits, procedure inlining may have negative effects. The primary disadvantage of inlining is that it increases code size, which might hurt instruction-cache performance by increasing instruction-cache miss rate [McF91a]. Besides, due to this code growth, the compilation time and the memory space consumption may become intolerable because some of the algorithms used for analysis have superlinear complexity. An alternative to inlining is to perform *interprocedural analysis* [RG89], which can be applied uniformly since it does not cause code expansion the way inlining does. However, it is difficult to model many important analyses in an interprocedural setting, and many of the analyses degrade significantly in the usual cases where not all program source is visible to the analyzer. And even if interprocedural analysis is performed, effective use of this information will most of the times require code expansion, since many of the code transformations enabled by an interprocedural analysis are impossible to safely express without some duplication of code in either the caller, the callee, or both.

Many research and production systems have been capable of performing procedure inlining. In most cases, they use some form of the following heuristics to select which procedure to inline [Muc97k]:

1. The size of the procedure body (the smaller the better).

2. How many calls there are to the procedure. If there is only one call, inlining it should almost always result in reducing execution time.

3. Whether the procedure is called frequently (e.g., inside a loop). If so, it is more likely to provide significant opportunities for other optimizations.

4. Whether a particular call site includes one or more constant-valued parameters. In this case, the inlined/cloned procedure body (i.e., the callee) is more likely to be optimized than if not.

McFarling investigated the impact of procedure inlining on instruction caches [McF91a]. Chang *et al.* [HC89b, CMCH92] and Ayers *et al.* [AGS97] demonstrate impressive performance improvement by aggressive inlining and cloning at high-level intermediate representation by using profile feedback and cross-module analysis. When inlining is carried out on executable programs after linking, the goal is to inline across module and library boundaries, such as in the Alto [AK00, MDWdB01] and PLTO [SDAL01] systems. As far as addressing the negative effects on instruction cache due to code growth is concerned, inlining is in this case more precise than inlining at the source level, since the optimizer has an accurate estimation of every procedure footprint and the size of the instruction cache.

2.3.1.5 Code positioning

The layout of instructions into memory is determined by the compiler [McF91b, RLPV01]. *Code positioning* (also known as *code layout*) determines not only the code page where an instruction is found, but also both the cache line (or set in a set associative cache) it will map to, and the behavior (taken or not taken) of conditional branches depending on the placement of their successor basic blocks. By mapping instructions appropriately in a different order, the compiler can have a direct impact on program performance.

There is an extensive body of work on code positioning optimizations [HC89a, McF89, PH90, TXD95, GBSC97, KK98, RLPN+99]. Ignoring minor differences between the different proposals, the main objective in all of them is to move infrequently executed code far from the frequently executed basic blocks (which are positioned close to each other), and *straighten* the code into linear sequences so that unconditional branches are avoided and conditional branches take the fall-through path as much as possible. As a result, a higher fraction of the instructions fetched from the instruction cache are actually executed, thus reducing instruction-cache conflicts. Code layout is generally applied to low-level or executable code, before applying code scheduling.

2.3.1.6 Code scheduling

Because many processors are pipelined and expose to the user at least some aspects of the dynamic hardware policy for executing instructions, it is essential that code for such machines be organized in such a way as to take best advantage of the pipeline or pipelines that are present in a particular architecture or implementation. *Code scheduling* is a common technique for reordering instructions to improve performance, which is among the most important optimizations for most programs on most machines [RF93].

Code scheduling is one of the last components of an optimizer to be executed when compiling a program, usually before code generation. This is the reason why there are no fundamental differences in performing code scheduling on intermediate or executable code. A widely used algorithm is called *list scheduling* [ACD74, SP89]. The algorithm operates on each basic block and can obtain significant improvements in code speed. The scheme has been also generalized to scheduling across basic block boundaries [Wal92, HMC+93, MR94], by considering a tree of blocks at once, which allows to move instructions from one block to another. Other sophisticated approaches can have large benefits for some types of programs and architectures, such as *trace scheduling* [Fis81, LFK+93], *balanced scheduling* [KE93, LE95], and others [RL92, EGK+94].

2.3.2 Profile-guided optimizations

Most of the early work on classical code optimizations is based upon a very simple performance model [ASU86a], since these optimizations are defined such that their application is always considered to have a positive impact on performance. Therefore, the focus of the research proposed over the past decades has been on both developing aggressive analysis techniques for uncovering as much opportunities for optimization as possible, and designing powerful

(a)
```
if ( ... )
{
    v = x * y;
}
if ( ... )
{
    w = x * y;
}
```

(b)
```
t = x * y;
if ( ... )
{
    v = t;
}
if ( ... )
{
    w = t;
}
```

Figure 2.3: Example of profile-guided optimization: (a) original code, (b) resulting code after redundancy removal.

program transformations to exploit most if not all of the uncovered opportunities, in order to speed up the execution time of programs.

While the simplicity of this conservative approach is attractive, it may fall into applying optimizations which might have an adverse impact of performance. Effectively, even although a static analysis algorithm may be successful in uncovering an optimization opportunity, the transformation phase still need to determine whether or not the optimization opportunity should be exploited. This drawback can be addressed by using a sophisticated performance model during the application of optimizations. In particular, a *cost-benefit analysis* of a program transformation can be carried out before actually applying the transformation itself.

To illustrate the above scenario, consider the example shown in Figure 2.3. The original code contains an optimization opportunity, since the expression `x * y` is computed twice when both conditionals evaluate to true. Moreover we can make this determination by statically analyzing the program. To optimize the uncover opportunity, we may wish to transform the code as shown in Figure 2.3b. This transformation removes the redundant evaluation of `x * y` for true evaluations of the conditionals. However, if we consider the complementary execution (i.e., false evaluations of the conditionals), we find that an evaluation of `x * y` is introduced in the transformed code where none was present prior to transformation. Therefore we can conclude that the optimization is useful only if evaluation of the first conditional to false is less frequent than the assignment to variable `w`. A simple cost-benefit analysis based upon expected execution frequencies of various statements in the program can be used to decide whether or not the transformation should be applied[4].

[4]While in the example both the cost and the benefit is being measured in terms of number of instructions or execution cycles, this may not always be the case. In some situations while the benefit may be measured in terms of anticipated reduction in the number of instructions, the cost may be measured in terms of code growth resulting from the transformation. This is because many optimizations essentially perform some form of code specialization, which results in code growth.

Profile-guided optimizations have received increased attention in recent years [GMZ02]. Before profile-guided optimizations can be carried out, an instrumented version of the program must be run on one or more representative inputs to collect profile data. This data is then used by the optimizing compiler to recompile the program generating optimized code. The example in Figure 2.3 illustrates that, by using this technique and given an adequate program's profile data, optimizing compilers are able to discover a broad category of optimization opportunities that may exists during program execution, so that they can now be exploited to aggressively optimize the program.

While this section is about static profile-guided optimization of programs, the concept of profile-guided optimization is also used by *dynamic optimizers* [DGR99, BDB00, CLCG00, MCE00, BGA03]. However, in contrast to the techniques described in this section, dynamic profiling and optimization must be extremely light weight. These techniques are beyond the scope of this work.

2.3.2.1 Types of profile information

Profiles provide summary information from past program runs that are used to guide program optimization [GMZ02]. The selection of representative inputs is important, since the optimizations are expected to be beneficial only if the profile data that is collected is relatively insensitive across a wide range of inputs on which the program is expected to be executed. Besides, different types of optimizations require different types of profiles. In this section we briefly discuss such types of profiles and the optimizations they are targeted.

The usual forms of profile are based on the program's *Control Flow Trace (CFT)*. This type of profiles captures a trace of the execution path taken by the program, which represents the order in which the basic blocks in the control-flow graph are visited. By examining a CFT we can compute the execution frequency of any given program subpath. As expected, CFTs can be extremely large in size and therefore representations that maintain CFTs in compressed form have been considered [Lar99, ZG01]. In practice, a number of approximations of CFT that directly measure the execution frequencies of selected program subpaths are used. These profiles differ in the degree of approximation involved and the cost for collecting them. The proposed approximations of control-flow profiles include the following:

Node profiles It provides the execution of the basic blocks in the CFG [Smi91]. For some optimizations such profiles are sufficient (e.g., making code placement decisions, as illustrated in Figure 2.3b).

Edge profiles It provides the execution frequency of each edge in the CFG [BMS98]. Edge profiles are superior to node profiles because basic block counts can be computed from edge counts, while the reverse does not always hold.

Path profiles It provides the execution frequencies of acyclic subpaths in the CFG such that they are acyclic and intraprocedural [BL96, BMS98]. Since a path is acyclic, it does not ever include a loop-back edge, and since it is intraprocedural, it terminates if an exit node of a procedure is reached. Path profiles are clearly superior to edge profiles, because path profiles can capture correlation across multiple conditional branches while edge profiles cannot do so.

To collect the profiles we must execute instrumented versions of the program. The instrumentation code that is introduced depends upon the type of profile being collected. Thus, the overhead of collecting edge profiles is comparable to the overhead of collecting node profiles, which is linear to the length of the program execution. However, several techniques can be employed to reduce this overhead [Sar89]. On the other hand, collecting path profiles is more expensive [BL96].

Other types of profile information have been used for a wide variety of profile-guided optimizations, such as *value profiles* [CFE97, MWD00, Chi01] and *address profiles* [Con97, CKJA98, RCT$^+$98].

2.3.2.2 Profile-guided classical optimizations

Simple optimization algorithms typically optimize statements that are determined to be optimizable under all conditions through static analysis of the program. On the other hand, more aggressive algorithms also optimize statements that are *conditionally optimizable* where the optimization opportunities are discovered either through static analysis or through profiling. As a consequence, often such algorithms involve replicating statements and creating unoptimized and optimized copies of them. Depending upon the conditions that hold, appropriate copy of the statement is executed. The above process is also commonly referred to as *code specialization*. In some optimizations, specialization leads to elimination of a copy (e.g., redundancy and dead code elimination) while during other optimizations specialization leads to simpler and more efficient code (e.g., strength reduction and constant folding).

The next section will provide a brief overview of code specialization transformations for eliminating programs redundancies. We also discuss the critical role that profiling plays in carrying out these type of optimizations.

2.3.3 Eliminating program redundancies

In general terms, redundant operations expose unnecessary recomputations at program run time of values that are already known, because they have already been computed in the program or because they can be computed at compile time. As a result, these redundancies expose optimization opportunities that can be potentially exploited by an optimizing compiler in order to remove them away.

In the long history of research and implementation of eliminating redundancies at compile time, four main types of transformations have been identified:

Code deletion The simplest form of removing a redundant statement is code deletion: if the value of a statement is previously computed along each incoming path, then the statement can simply be removed. To verify that the redundancy exists along all paths, the optimization can be restricted to basic blocks, as in *common subexpression elimination* [ASU86a] or *value numbering* [Muc97l]. For applying deletion globally (across multiple basic blocks), data-flow analysis is applied to confirm that the value to be removed is available along all paths.

Code motion Deletion is impossible when statement is redundant along a strict subset of all incoming paths. Code motion is a technique that hoists the partially redundant statement so that it is removed from paths on which it is redundant. Effectively, hoisting introduces compensation code on non-redundant paths, changing partial redundancy into full redundancy, which enables deleting the statement from its original position. *Loop invariant code motion* is the simplest form of such motion transformation [Muc97k]. Morel and Renvoise generalized it to arbitrary control-flow graphs by formulating the code motion problem as a bidirectional data-flow analysis [MR79], while Knoop *et al.* found an unidirectional formulation by decomposing the bidirectional problem into two unidirectional problems: availability and anticipability (also called *very busy expressions*) [KRS94a].

Control-flow restructuring The necessary code motion may be blocked when it would change program semantics or impair the program for certain inputs. When code motion fails to eliminate all partial redundancies, control-flow restructuring can be applied. Restructuring is based on separating the optimizable paths from the unoptimizable paths, which is accomplished by duplicating all statements along the path that needs to be separated. A simple form of restructuring is *tail duplication* [HMC+93], which separates frequently executed paths to improve scheduling by separating control-flow merge points. Restructuring is also necessary when redundant operations are unhoistable, such as branch instructions [MW92, MW95, BGS97].

Control speculation This form of transformation inserts computations onto paths that did not compute them in the unoptimized program [HH97b, GBF98, LCK+98]. As a result, some paths are optimized and some are impaired. To control the impairment, a run-time program profile is often used.

Other kind of redundancy elimination optimizations are also built on top of one or more of these transformations. As an example, *partial dead code elimination* eliminates statements that compute a value that will not be used in the remainder of the program, by using code motion alone [KRS94b]. Dead values that cannot be removed with code motion must be eliminated through restructuring [BG97].

Clearly, the four types of transformation differ in their power and cost [BGS98]. Deletion is only applicable on fully redundant operations, and hence is not suitable for partial redundancies. While code motion is efficient in that it does not increase code size, it is less powerful than restructuring, which can eliminate all redundancies but may incur in significant code explosion. Finally, control speculation does not remove all redundancies and it impairs some paths, but it does not introduce code duplication at all.

We next review prior work related to eliminating program redundancies by classifying it according to the type of redundancy it tries to address. Such classification leads to a better understanding of the underlying problems we deal in this thesis.

2.3.3.1 Partial redundancy elimination

By attempting to remove redundancies that occur only on some control-flow paths of a program, *Partial Redundancy Elimination (PRE)* [MR79, KRS94a, CCK+97] subsumes various well-known *ad hoc* code motion optimizations, such as common subexpression elimination and loop invariant code motion [ASU86a]. Horspool and Ho [HH97b] described a speculative formulation of PRE based on a cost-benefit of the flow graph, by using edge profiles. This approach has been generalized to provide optimal solutions both for time [CX03] and space [SHK04]. Gupta, Berson and Fang [GBF98] extended the general PRE algorithm by using path profiles. Finally, Bodík, Gupta and Soffa [BGS98] developed a profile-driven PRE approach using path profiles and control-flow restructuring, which is complete. However, as they replicate regions of code when needed, some code growth also results.

The problem of path-sensitive redundancies has been described by Bodík and Anik [BA98, Bod99]. They propose a new representation called *Value Name Graph (VNG)* to be used for general path-sensitive optimizations. However, it is not obvious that using the VNG for optimizing large programs would scale up to problems of this size.

2.3.3.2 Register promotion

Eliminating memory redundancies is a form of PRE, where the expressions to be considered for removing are only memory operations. In this way, *register promotion* allows scalar values to be allocated to registers for regions or their lifetime, where the compiler can prove that there are no aliases for the value. Promotion carries out elimination of both redundant loads and stores [CCK90]. Cooper and Lu [CL97] examined promotion over loop regions. Their results indicate that the main benefit of promotion comes from removing store operations. Lo *et al.* [LCK+98] use a variant of SSA-PRE to remove unnecessary loads and stores over any program region. However, they do not consider the effect of spilling because they simulate with an infinite symbolic register set before register allocation. They also counted the improvement just by comparing the total number of load and store instructions.

Postiff, Greene and Mudge [PGM00b] presented a register promotion algorithm at link time, although their algorithm does not make use of any PRE approach at all. They also present numbers for large register files, but the gain in this case seems to come from several *ad hoc* techniques for promoting global and constant values into a dedicated subset of the register file. Finally, Bodík, Gupta and Soffa [BGS99] developed a load redundancy analysis and designed a method for evaluating its precision by using the VNG. However, their paper is only focused in the analysis, and they do not perform any elimination of redundant load instructions at all.

2.3.3.3 Elimination of conditional branches

Removing conditional branch redundancies can be seen as a particular case of partial redundancy elimination. However, the code motion techniques useful for PRE of assignments do not suffice for removing conditional branches. To eliminate a conditional, *control-flow graph restructuring* is usually required.

The simplest form of branch elimination is loop unrolling [Muc97j], in which instances of back-edge branches are removed by replicating the body of the loop. More sophisticated techniques examine control and data flow simultaneously to identify correlation among branches. An algorithm for intraprocedural restructuring was first proposed by Mueller and Whalley [MW95], although their technique was mostly focused on eliminating conditionals within loops. A more general approach based upon interprocedural demand driven analysis as well as profile-guided control-flow restructuring was given by Bodík, Gupta and Soffa [BGS97]. However, a lot of interprocedural redundancy they remove could be also eliminated by simply applying constant propagation after function inlining.

2.4 Binary optimizers

A *binary rewriting system* transforms a binary program into a different but functionally equivalent program. A *binary optimizer* is a binary rewriting system that modifies an object program to improve some aspect of its behavior (e.g., execution time, code size, power consumption, etc.) *after* it is compiled (see Section 1.1.2). Traditionally, it is the task of the compiler or assembler to generate object code and it seems cumbersome to change object code once it has been produced. Nevertheless, the number of applications where object code modification is successfully employed grows rapidly, such as *binary translation* [SCK$^+$93, HH97a, CE00, AKS00], *program profiling* [LB94, SE94], or *debugging/sandboxing* [HJ92, WLAG93].

2.4.1 Advantages of performing binary optimizations

The optimization of a binary may occur at a very late stage *during* linking (i.e., link time) or *after* linking (i.e., post-link time). Both approaches are quite similar: integrating optimizations within the linker will simplify parsing of the code and might give access to slightly more information about the source program, while changing object code after linking provides a very clean separation of responsibilities and does not require access to potentially proprietary linker source[5]. In any case, performing optimizations at the object code level has the following advantages:

Independency of the source language/compiler Working with object code makes optimizations essentially compiler and language independent, similar to a common back end coupled with several compiler front ends [ASU86b]. However, it may be necessary to recognize certain compiler and/or language specific sequences at the object code level and treat them specially, in order to improve the effectiveness of the optimizations.

Easy addition of optimizations to the compilation process Usually the source code of a compiler is not available for modification, or the documentation is so poor that adding new optimizations might be difficult. Hence, it is very popular to try out new optimizations in a simple and well documented compiler like `lcc` [FH95] whose source is publicly available. However, it is questionable whether results obtained in this way will transfer to a production quality compiler. Applying optimizations at link time allow us to essentially add optimizations to the best available compiler without modifying it.

Supplement optimizations not well performed by the compiler Some optimizations cannot be easily performed at compile time, specially those ones that rely on an accurate estimation of the number of low-level machine instructions. At link time, we can easily obtain this type of information not available at compile time, so that optimizations can be optimally applied.

Availability of the entire program Large programs tend to be compiled using separate compilation, that is, one or a few files at a time. Therefore, the compiler does not have the opportunity to optimize the program as a whole, even when performing sophisticated interprocedural analyses and transformations. As a consequence, link-time or executable code optimizers have the attraction of being able to work when the *entire* program is available for inspection. This is even valid when source code of programs is unavailable, such as in old legacy software or library code. Optimizing the object code appears to be the only way to improve performance of these programs.

[5]For the rest of this work we will not distinguish between these two approaches.

Easy use of profile information Generating profile information by instrumenting object code is very popular and also fairly easy (see Section 2.3.2.1). The problem is to exploit this information in an optimizing compiler. There is an *impedance mismatch* between the information provided by the object code level profiling and the source level compiler, similar to the one found in a source level debugger. This is because the low-level profile information needs to be back mapped to source code, which is a non-trivial problem specially when code is highly optimized. When optimizing at the object code level, on the contrary, this mapping is one-to-one and does not present any problems.

The cost metric that classical optimizations try to reduce is execution time, although they often also reduce code size as a side effect. However, in recent years there has been an increasing trend towards reducing space, since computers are being incorporated in devices and embedded systems where the amount of memory is limited. In this case, we can reduce code and data size by using special compression techniques [EEF$^+$97, DEMdS00, DE02]. We will not consider these optimization techniques in this work.

2.4.2 Related work

We next describe the most relevant projects in the area of object code modification for optimizing binaries.

OM The *OM optimizer* is a tool that improves the performance of Unix applications on Compaq/Alpha processors, that can also make use of profile information [SW92]. It uses the OM's object code modification framework, which is a library similar to EEL [LS95] that tries to hide much of the complexity of editing object files. The framework is dedicated to the exploration of techniques for the modification of binary machine code, such as program instrumentation [SE94].

OM was designed as a separate pass after linking, but it relies on the linker to provide additional information not found in the executable. Internally, it translates instruction to a *Register Transfer Language (RTL)*, which can be manipulated and translated back to machine instructions. Another of the design goals for the OM optimizer was to make it fairly light-weight. Thus, it does not perform many optimizations, and the ones it does perform are restricted to intraprocedural optimizations that do not consume a lot of resources, such as faster global variable access [SW94], unreachable code removal, procedure inlining, and basic block reordering.

Etch Developed for Intel/x86 platforms running the Windows/NT operating system, *Etch* is a program performance evaluation and optimization tool that allows to annotate existing binaries with arbitrary instructions (for example, to trace, or perform coverage analysis), or to rewrite an existing binary so that it executes more efficiently [RVL+97]. To instrument a program, Etch is invoked with a *Dynamic Linked Library (DLL)* containing the analysis code in the form of call-back functions that are invoked by Etch to modify the executable. Those functions can in turn call the Etch interface to perform the actual instrumentation.

Etch also provides facilities for rewriting an executable in order to improve its performance. For example, the instrumentation phase, rather than adding new instructions, can direct Etch to write the executable out according to a different code layout optimized for cache behavior.

Spike This tool is an executable optimizer for Compaq/Alpha binaries originally intended for Windows/NT applications [CGLR97, CGL97], which was later modified for Tru64 Unix applications [FLM+01]. It can handle executables and shared objects, performing both instrumentation and optimization. The instrumentation part is a *Pixie* adaptation which provides basic block and control-flow edge execution frequencies [Smi91], although it is also able to use estimated counts collected with the DCPI statistical profiler [ABD+97]. Profile data is then used to improve a variety of analyses and optimizations, such as register liveness analysis [Goo97], hot-cold optimizations [CL96], and register allocation [Muc97h], as well as reorganizing executables to improve cache locality [RBG+01].

Recently, a similar tool called *Ispike* has been developed for optimization of Intel/IPF executables on the Linux operating system [LMP+04]. Besides the standard optimizations, it implements a number of key optimizations targeting memory latency, including code layout, instruction prefetching, data layout and data prefetching. They are driven by several types of profile information collected via the Itanium performance monitoring hardware [ME02].

Alto Within the scope of the University of Arizona's SOLAR project, *Alto* is a post-link time optimization system for Tru64 Unix applications on Compaq/Alpha processors [MDWdB01]. Alto implements a large number of profile-guided optimizations, (e.g., code layout, inlining, etc.) but also a wide variety of novel whole-program data-flow analyses and code optimization techniques well suited for optimizing programs at link time [Mut98, DMW98, Mut99]. As an example, it includes a novel optimization for low-level value-based program specialization [MWD00]. Alto is the framework we have used to implement the proposals presented in this thesis, as we will see in Section 3.2.1.

SOLAR also includes other link-time optimization systems. These optimizers are very similar to Alto, and thus share most of its features. *PLTO* is targeted at optimizing Intel/x86 applications on Linux [SDAL01]. It includes a novel stack analysis for optimizing parameter passing [LDAS04]. *ILTO* is an optimization system for Intel/IPF executables on Linux [SDA02]. In this case, the main contribution comes from the implemented techniques for reverse engineering Itanium applications [SDA03b, SDA03a].

Squeeze This tool (and its C++ evolution, *Squeeze++*), is a post link-time compaction tool based on Alto whose main goal is to devise techniques for reducing the memory footprint of Tru64 Unix executables for Compaq/Alpha processors [dSdBdBD01]. Besides applying whole-program optimizations to reduce execution time of programs, Squeeze also tries to reuse code and data on several levels of granularity to reduce program size. In particular, *code factoring* refers to a variety of techniques to identify and "factor out" repeated instructions sequences [DEMdS00, dSVdBdB03]. Squeeze++ also applies some code reuse techniques specifically targeted at C++ language features, such as inheritance and the use of templates [dSdBdB02].

The Squeeze team has recently developed a retargetable framework for link-time code editing called *Diablo* [dBKC$^+$03, dBdSvP04]. Diablo can be used to optimize programs for code size or speed, to add instrumentation code or to get better understanding about a program at the binary level.

Chapter 3

Experimental environment

This chapter presents the experimental environment on which this thesis has been developed. We first give an overview of the target platform used for experiments, and also introduce our experimental framework at compilation and simulation levels. We also describe the benchmark suite and the methodology used for evaluating the benefits of our proposals.

3.1 Target environment

This section describes the environment we choose to develop the research presented in this thesis. We also present some of the tools employed in this work for generating the different sets of programs that will be used for evaluation.

3.1.1 Target platform

The execution environment chosen as our target platform is an AlphaServer GS–140 running the Compaq/Tru64 UNIX operating system (version 5.1). This server is equipped with a 525MHz Alpha EV6 21264 processor [KMW98]. The main processor characteristics can be seen in Table 3.1. There are several reasons why we have chosen this platform as our experimental environment:

- The Compaq/Alpha 21264 processor is a superscalar out-of-order microarchitecture that was designed to operate at frequencies higher than other processors of its generation. Besides, its microarchitecture specification has been published in much more detail than chips from other vendors [KMW98, Kes99, Com99a].

- Binary optimizations are directly applied to the *Instruction Set Architecture (ISA)* level. Although working at the ISA level might seem to be very non-portable, the Compaq/Alpha processor has a very generic RISC ISA, which is in fact very similar to the low-level intermediate representation used in many compilers [ASU86d, Mah92, Muc97e, Mor98]. Then, the proposals presented in this work should be easily transferable to other RISC architectures.

- The platform chosen is stronger than other platforms in development support. Besides having different compiling and debugging tools, it provides a wide variety of tools available for the propose of program instrumentation [Smi91, SE94], binary optimization [MDWdB01], and performance simulation [ALE02], which are targeted to the Alpha environment.

As a result, this platform allows us easily both implement our low-level compiler proposals within a binary optimizer, and obtain a wide variety of measurements for evaluation.

3.1.2 Compilation environment

In this work we are interested in evaluating new compiler optimizations at the binary level. Therefore, we need to generate an appropriate set of benchmarks so that we can measure the effects of applying such optimizations to them. We use in this work two different sets of programs:

Parameter	**Value**
Instruction set	64-bit load/store RISC little-endian architecture. 32-bit instructions.
Pipeline	Seven stages plus a variable number of execute stages.
Fetch width	4 instructions per cycle (128-bit packed).
L1 I-cache	64Kb, 2-way set-associative, virtually addressed cache with 64-byte line, 1-cycle hit latency.
Branch predictor	Combined branch predictor. Local: 1K 10-bit local histories that access 1K 3-bit saturating counters. Global: 12-bit global history register that access 4K 2-bit saturating counters. Choice: a predictor that selects the local or global predictor, with 4K 2-bit counters.
Decode/Commit width	4 instructions per cycle.
Register file	Two separate integer and floating point register files, 31 register per file. Register renaming, 80 integer and 72 floating point physical registers.
Instruction window	Up to 80 instructions in-flight.
Issue mechanism	Speculative execution. Up to 4 integer and 2 floating point instructions per cycle. 20-entry integer and 15-entry floating point queues.
Execution units	4 integer, 2 floating point pipeline units
Load/store queue (LSQ)	32-entry load and store queues. Stores may bypass values to later loads.
L1 D-cache	64Kb, 2-way set-associative, virtually indexed, physically tagged, dual-read-ported, write-back cache with 64-byte line, 3-cycle hit latency.
L2 unified I/D-cache	2Mb, 4-way set-associative, physically indexed cache with 128-bit bus to L1, 16 bytes to main memory, 16 cycles first chunk, 2 cycles interchunk, 12-cycle hit latency.
Instruction TLB	128-entry fully-associative, 8Kb page, 30-cycle miss penalty.
Data TLB	128-entry fully-associative, 8Kb page, 30-cycle miss penalty.
Main memory	1Gb, 128-bit bus to L2, 80-cycle hit latency.

Table 3.1: Compaq/Alpha EV6 21264 processor characteristics.

Baseline set This set is obtained from compiling our programs on the native environment and getting adequate profile data. Then, a binary optimizer is used to read the resulting binaries and their profiles, and apply profile-guided optimizations to obtain highly optimized programs.

Optimized set The other application set involved is obtained by following the same process for obtaining the baseline set. This time, though, we include within the binary optimizer the particular analysis or optimization in which we are interested.

It is critical in this work to compare the effectiveness of our proposals against state-of-the-art optimized machine code. This is the reason why sophisticated profile-guided binary optimizations are actually applied in the baseline set, as we will see in Section 3.2.

We next describe in some depth the steps we have followed and the tools involved in the compilation process, which defines our compilation environment.

3.1.2.1 Native compilation

For generating our initial set of executable programs from source code applications, we used the Tru64 C compiler (version 6.3) on the environment presented in Section 3.1.1. The compiler was invoked with full optimizations and special linker options to produce statically linked executables and to retain relocation information, as follows:

```
cc -O4 -arch ev6 -non_shared -Wl,-r -Wl,-d -Wl,-z
```

We used statically linked executables since most of the object code modification tools are not able to process non-statically linked executables with shared libraries. Furthermore, relocation information is also needed for our post-link time processing tools. Otherwise, any instruction must be considered the target of an indirect jump [SW92].

3.1.2.2 Getting profile information

Once we have our set of original binaries, we are also interested in obtaining adequate profile information so that profile-guided optimizations can be applied at the binary level.

Our programs were instrumented using *Pixie* [Smi91], and then executed using representative inputs to obtain a basic block execution frequency profile. We will show in Section 3.2.1.2 how different types of control-flow profiles can be derived from basic block counts.

3.1.3 Execution environment

Since the proposals we present in this work will be implemented within a compiler or binary optimizer, and they do not involve any hardware modification at all, the most interesting measure to evaluate their effectiveness is to report actual execution times. Therefore, we run the obtained benchmark sets on the target environment presented in Section 3.1.1, recording in each case the smallest of seven runs of programs running in single-user mode.

3.2 Experimental framework

In this section, we describe the experimental framework used for implementing and evaluating the ideas presented in the following chapters.

3.2.1 Binary optimization environment

The final step in our compilation chain must read the original executable programs produced by the native linker (as described in Section 3.1.2.1) together with their corresponding profiles, in order to apply additional analyses and profile-guided optimizations. The goal is to optimize the programs for performance at the binary level as much as possible.

We have chosen the *Alto* link-time optimizer [MDWdB01] (see Section 2.4) to perform the optimization task at the binary code level. The resulting executable programs will be the baseline for evaluating the proposals presented in all further chapters, since this baseline can be considered state-of-the-art optimized machine code. Furthermore, as the Alto source code is available for modification, this tool will also be the binary optimization tool where our algorithms will be implemented. Therefore, the algorithms we propose in this work will be easily integrated with the rest of optimizations carried out by Alto. This approach ensures not only exposing benefits coming from the new optimizations themselves, but also enhancing the effect of the rest of Alto optimizations.

3.2.1.1 Optimization phases

After reading in the executable file (containing relocation information for its objects) and transforming it into an intermediate form, Alto optimizes a given binary in several phases. We next outline the overall structure of the optimizer [MDWdB01]:

Base optimizations First, a suite of simple optimizations is carried out iteratively by Alto. This suite includes most of the classical compiler optimizations (see Section 2.3.1), such as constant propagation/folding, dead/unreachable code elimination, etc.

The base optimizations suite is iterated until either a fix point is reached or a maximum iteration count is exceeded. As these optimizations are relatively cost-effective, they can be applied several times during the optimization process without significantly increasing total compilation time.

One-time optimizations In this phase, Alto performs optimizations that should only be done once because either (a) the optimization may require costly analyses, or (b) repeating the optimization might have undesirable side effects, or (c) repeating the optimization will not give any additional benefit. The most important optimization in this phase is *procedure inlining* (see Section 2.3.1.4). Inlining at this level presents a good opportunity to remove calling convention overhead that may not have been exploited when compiling source code.

Late optimizations After all optimizations have been executed, Alto performs code positioning to improve instruction-cache usage. The approach used is a variation of the Pettis and Hansen algorithm [PH90]. Finally, instruction scheduling is applied to improve program performance.

All optimizations are supported by the corresponding needed analyses, such as those presented in Section 2.2. Finally, intermediate representation is transformed back into object code, and then written out.

3.2.1.2 Enhancing optimizer capabilities

Beyond the proposals exposed in the next chapters, we have either added or improved some of the Alto capabilities and optimizations. The reason is that some optimizations were not targeted at the platform presented in Section 3.1.1, but also because some capabilities and optimizations were not powerful enough to expose the desired redundancy existing in programs. We next describe these modifications in some depth:

Copy propagation Eliminating register copies in Alto was limited to recognize and remove some common patterns within basic blocks [Mut99]. However, we found that performing copy propagation beyond EBB boundaries becomes significantly important after several transformations, such as procedure inlining and redundancy elimination (see Chapter 5). We therefore implemented a general intraprocedural copy propagation algorithm [Muc97l] as part of the Alto base optimizations.

Procedure inlining As we have shown in Section 2.3.1.4, applying procedure inlining at the machine code level yields some important benefits. Furthermore, it gives intraprocedural algorithms an interprocedural behavior when they are applied after inlining. However, despite its benefits, inlining may have negative effects due to code explosion. The technique used by Alto to mitigate this effect is to inline functions only if at least one of the following conditions hold:

1. The callee is *small enough* that the calling and return sequences together are longer than its body.
2. The call site under consideration is the only call site for the callee function.
3. The call site is *hot enough* (i.e., has a sufficiently high execution count), and the estimate cache footprint of the resulting code after inlining does not exceed the size of the instruction cache.

The first two conditions are always beneficial since there is no increase in code size. The reason for the last condition is that inlining without attention to cache behavior can have a significant negative impact on program performance. Therefore, there is a trade off between reducing call overhead and increasing code growth. To address this problem, Alto applies some heuristics based on an execution frequency profile [MDWdB01].

We have improved the current Alto inlining by sorting the inlineable functions so that frequently call sites calling smaller functions are considered first for inlining. Besides, we have included a bail out condition to avoid total code grow beyond a certain threshold. More sophisticated strategies are possible [McF91a, CMCH92, AGS97], but these have not been considered for this thesis.

Code scheduling The instruction scheduler implemented in Alto is a slight extension of a regular list scheduler that works on extended basic blocks [MDWdB01]. It is however subject to the restriction that the basic blocks constituting the EBB must be consecutive in the code layout, which allows instructions to move across nodes if this motion preserves correctness of the program. As a result, they achieve an effect very similar to trace scheduling [Fis81].

The Alto scheduler is targeted to an Alpha 21164 EV5 pipeline that does not match with our target platform, which belongs to the next generation of Alpha processors. However, the need for scheduling machine code for the Alpha EV6 is somehow less important, since it is a dynamically scheduled processor [KMW98]. Thus, we have only updated the corresponding instruction latencies from EV5 to EV6 information. We have also implemented a previous optimization for renaming registers [Muc97j], in order to remove unnecessary false register dependencies so that flexibility available to code scheduling can be increased.

Profiling support Our programs were instrumented using Pixie [Smi91] to obtain execution frequency profiles, as we have shown in Section 3.1.2.2. Furthermore, we need different types of control-flow profiles (see Section 2.3.2.1), since some of the analyses and optimizations we present need some form of edge frequency counts, instead of simply basic block counts. However, the Pixie instrumentation tool only provides basic block execution counting. Although it is widely known that block counts can be derived form edge counts and the converse does not hold [BMS98], edges whose counts cannot be determined from block counts are usually fewer than 1%. Therefore, we have implemented in Alto a variation of the algorithm from Tamches and Miller [TM01] for deriving edge counts from Pixie profile data.

Even though the effects of applying the optimizations and enhancements we have described have a positive impact on program performance, the main goal has been to expose binary redundancy as much as possible, in order to maximize the proposals presented in the following chapters. Eliminating binary redundancy is then addressed in Alto within the two optimization phases (see Section 3.2.1.1):

Base optimizations We include in this suite the most light weight algorithms we propose.

One-time optimizations After the one-time optimizations implemented by Alto, we apply at this point the most expensive versions of our optimization algorithms, coupled with the needed space- and time-intensive data-flow analyses. As we will see, the majority of algorithms we propose in the following chapters are intraprocedural (i.e., they are applied at procedure boundaries). However, Alto has previously applied procedure inlining, which will give to our algorithms an interprocedural behavior.

To catch any new opportunities opened up by the expensive one-time optimizations, including the ones we will developed in this work, an additional round of base optimizations is performed before and after the one-time optimization version of the particular algorithm we are proposing.

3.2.1.3 Execution threshold

Some of the analyses and profile-guided optimizations applied by Alto, and also some of the algorithms we proposed in this document, use profile information for having accurate execution frequencies of the different parts of the program, but also for grouping program entities into different *execution sets*. The reason is that, since they are expensive algorithms in terms of compilation time and/or memory requirements, they need to be applied to only a *hot set* of the program to keep these requirements under control. For example, the hot set of basic blocks consists of the most frequently executed blocks in the program (according to some threshold ϕ, as we discuss below). The notion of hot sets may also be applied to other program entities, such as control-flow edges or programs paths.

As far as basic blocks is concerned, we need to determine the set of blocks that are executed "sufficiently frequently". Thus, given a value ϕ in the interval $(0, 1]$, we determine the largest execution frequency threshold N such that, by considering only those basic blocks that have execution frequency at least N, we are able to account for at least a fraction ϕ of the total number of instructions executed by the program (as indicated by its basic block execution profile). In other words, basic blocks are sorted according to their relative execution frequency so that a particular algorithm only considers instructions within basic blocks that have an execution frequency larger than $1-\phi$. Those basic blocks are then said to be *hot* with respect to the threshold ϕ. By using the same parameters, we can generalize hot sets to almost any program entity. Therefore, an edge is consider to be a *hot edge* with respect to the threshold ϕ when its execution count is at least N. Similarly, a *hot path* in a program is a path that only contains hot edges.

The value N, and therefore the corresponding hot set, obviously depends on the threshold ϕ. For example, given $\phi = 0.95$, the hot basic blocks of a program consist of those that account for at least 95% of the instructions executed at run time. On the other hand, a value of $\phi = 1.0$ will consider every basic block to be hot. Therefore, large ϕ values also cause a large increase in optimization time and possibly memory requirements.

We have used different values of ϕ along the different proposals of this work, which have been determined via empirical tuning[1]. However, we have observed that the final results are not very sensitive to the final value of ϕ in every case.

[1]We provide in every chapter the different values of ϕ used in every case.

3.2.2 Simulation environment

It is often the case that obtaining simple machine runs, as we do in Section 3.1.3, does not give us detailed information about the behavior of the algorithms under evaluation. For example, obtaining the exact number of instructions executed, the dynamic breakdowns based on instruction types, or even detailed information about hardware events (e.g., cache misses) is not possible from simple program runs. To perform this task, we could have used profiling tools that obtain such information from hardware performance counters [ABD$^+$97, Hun00]. However, these tools are often not accurate enough for our goals, since the obtained measurements come not only from the measured program, but from parts of the operating system as well. Besides, not every data can always be inferred in the collection process.

The approach we have chosen to address the above problem is *microprocessor simulation* [MAA$^+$02]. Simulation has been used over the past decades for enabling exploration of design alternatives for future high-performance computers, without having the cost of actual building the whole hardware system. Furthermore, simulation usually provides more detailed statistics collection than real hardware, thus obtaining meaningful results of program performance on complex architectures.

While the architecture research community relies heavily on simulators [RBDH97, OG98, SL98, LCA01, ALE02, EAB$^+$02, HPRA02, MCE$^+$02], the main problem of using this technique comes from the fact that simulators often are highly inaccurate due to abstraction, specification, or modeling errors. Consequently, researchers may draw wrong conclusions when evaluating a new proposal. The solution to this problem is to *validate* the performance model against a baseline context, or even better, against real hardware [BS98]. This is not always possible when evaluating new hardware proposals, but should be the case for new proposals that provide no hardware modifications.

The simulator we use in this work to measure the different aspects of the proposed algorithms is the `sim-alpha` simulator [DBK01], which faithfully models a Compaq/Alpha 21264 configuration that matches our target environment presented in Section 3.1.1. Besides, it has been validated against a Compaq workstation obtaining less than 2% average error, which make us feel confident about the validity of the conclusions derived from our simulation runs.

The `sim-alpha` simulator is based on the *SimpleScalar* toolset [ALE02], which provides an infrastructure for simulation and architectural modeling. The toolset includes several sample models suitable for a variety of common architectural analysis tasks. Thus, among other included simulators, they provide `sim-safe` (a minimal instruction set emulator), `sim-profile` (a dynamic program analyzer), and `sim-outorder` (a detailed microarchitecture timing model). We have used the SimpleScalar simulators for collecting information that does not depend on the processor architecture but relies somehow on the instruction set, such as measuring number of executed instructions, number of memory references, etc.

Benchmark	**Description**	**Type**	**Input**	**Set**
`099.go`	Board game	Profiling Execution Simulation	50 9 50 21 50 10	Train Reference —
`124.m88ksim`	Motorola 88K simulator	Profiling Execution Simulation	dcrand.lit {dcrand,dhry}.lit dcrand.lit	Train Reference Train
`126.gcc`	GNU C compiler	Profiling Execution Simulation	amptjp.i cp-decl.i gcc.i	Train Reference Reference
`129.compress`	Lempel-Zip encoder/decoder	Profiling Execution Simulation	10000 q 2231 14000000 e 2231 50000 e 2231	Train Reference —
`130.li`	Lisp interpreter	Profiling Execution Simulation	boyer.lsp {*}.lsp boyer.lsp	Train Reference Train
`132.ijpeg`	JPEG encoder/decoder	Profiling Execution Simulation	specmun.ppm vigo.ppm specmun.ppm	Train Reference Train
`134.perl`	Perl interpreter	Profiling Execution Simulation	primes.in primes.in primes.in (51 lines)	Train Reference Reference
`147.vortex`	Database application	Profiling Execution Simulation	persons.250 persons.1k persons.250	Train Reference Train

Table 3.2: SPEC95 integer benchmark suite and their inputs used for profiling, execution, and simulation.

3.3 Benchmark suite

A very important choice in this study is the set of programs to be analyzed. The SPEC95 benchmark suite was released on August 1995 by the *Standard Performance Evaluation Corporation (SPEC)* [Rei95, DR95]. SPEC represents a worldwide standard for measuring and comparing computer performance across different hardware platforms and optimizing compilers [Wei97]. SPEC95 was developed by SPEC's *Open Systems Group (OSG)*, which includes more than 30 leading computer vendors, publishers and consultants throughout the world.

The benchmarks we used for evaluating our proposals were the eight programs in the SPECint95 benchmark set. Table 3.2 shows a brief description of these programs. We have chosen only integer applications because, besides being widely used, these are the types of applications that are most difficult to be optimized by a traditional optimizing compiler. This is the reason why binary optimization tools usually obtain better results on integer programs compared to those obtained for floating point programs.

The SPEC benchmarks come with a number of different input data sets:

- The *test* input set is a short dataset to ensure that the benchmark is working correctly.
- The *train* input set is often used to get profile information after program instrumentation, in order to use it as feedback data for profile-driven compiler optimizations.
- Finally, the *reference* input set is the official input that is used for generating the official performance scores published on the SPEC web site.

In this work, we have used the different input sets for different purposes, as shown in Table 3.2. The *train* and *reference* input sets are the ones we used for collecting profiles and for reporting actual execution times, respectively (see Section 3.1.2.2 and Section 3.1.3). When simulating benchmarks, though, the use of these input sets is either too short to give representative results, or too long so simulation is prohibitive. There are methods described in the literature for simulating only representative portion of the programs [KFML00, LS00, NS01, SPC01, SPHC02]. However, the approach we followed was using variants of the SPEC input set to keep simulation time down to a manageable value.

A more recent version of SPEC benchmarks exists under the name of SPEC2000, but we have used SPEC95 for several reasons. First, when the work included in this document was started, SPEC2000 was not available to the community. Thus, when this version appeared, we thought that continuing with SPEC95 was appropriate to normalize results and understand them better. The second reason relies on the fact SPEC95 fits better to the generation of our target platform (see Section 3.1). Finally, the SPEC2000 generally suffers from worse cache-miss ratio than the SPEC95 programs. As a significant part of our work focuses on on-chip memory behavior, we thought that to use SPEC95 was more representative.

3.3.1 SPECint95 characterization

We next describe the process we follow to characterize the programs of the SPECint95 benchmark suite. This section will give us detailed information of the effects of our compilation framework on program optimization, but will also show the methodology we use for the rest of this document when evaluating the benefits of our proposals.

The process used is the following. We first compiled the SPECint95 programs in the compilation environment described in Section 3.1.2.1. The obtained programs are considered the baseline in this section, and we will refer to them as the *Original* benchmark set. These programs were then instrumented to get profile information, and then re-optimized using our enhanced version of Alto (see Section 3.1.2.2 and Section 3.2.1, respectively). Two new benchmark sets were generated (i.e., *Alto/Base* and *Alto/Inline*), by running Alto with and without applying procedure inlining with execution threshold $\phi = 0.75$ (see Section 3.2.1.3).

Benchmark set	Description
Original	Programs compiled with native C compiler.
Alto/Base	Original binaries optimized using Alto, with profile information.
Alto/Inline	Original binaries optimized using Alto, with profile information, including procedure inlining ($\phi = 0.75$).

Table 3.3: Description of the different benchmark sets under evaluation.

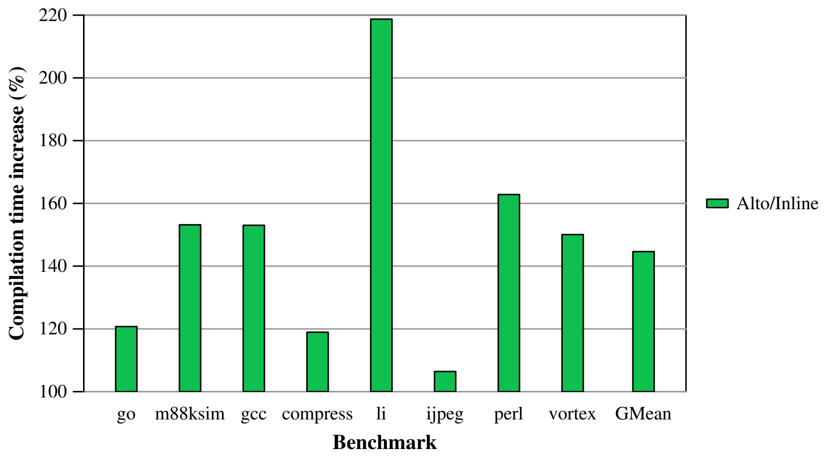


Figure 3.1: Effect of applying inlining in Alto compilation time, for the SPECint95 programs. The baseline is binaries after being optimized by Alto, without applying inlining (i.e., *Alto/Inline* binaries with respect to *Alto/Base* binaries).

The reason why we generate two additional benchmark sets is that we are specially interested on the effects of inlining in our optimization chain. A description of the benchmark sets we have generated is shown in Table 3.3.

3.3.1.1 Compilation time

The first data we report is the compilation time comparison of Alto when procedure inlining either is or is not enabled (i.e., *Alto/Inline* binaries against *Alto/Base* binaries[2]). Figure 3.1 shows that inlining increases compilation time in less than 60% for almost all programs, which is a very moderate growing percentage. The most interesting cases are programs `ijpeg` and `li`. Program `ijpeg` had few opportunities for inlining, since most of its important call sites invoke procedures by using indirect calls. On the contrary, program `li` suffers the opposite effect: there is a high number of opportunities for inlining, and the inlined procedures significantly increase the final program size (as we will see in the next section). As a result, total compilation time is increased.

[2]The *Original* benchmark set is not considered here since native compilation time is significantly lower.

(a) SPECint95 characteristics for *Original* binaries

Benchmark	Functions	Nodes	Edges	Instructions
`099.go`	691 (1.00)	16313 (1.00)	30437 (1.00)	81154 (1.00)
`124.m88ksim`	609 (1.00)	11712 (1.00)	22347 (1.00)	50557 (1.00)
`126.gcc`	2244 (1.00)	76666 (1.00)	157190 (1.00)	323415 (1.00)
`129.compress`	334 (1.00)	5843 (1.00)	10583 (1.00)	22158 (1.00)
`130.li`	711 (1.00)	9907 (1.00)	18989 (1.00)	39293 (1.00)
`132.ijpeg`	742 (1.00)	12003 (1.00)	22282 (1.00)	60848 (1.00)
`134.perl`	713 (1.00)	23071 (1.00)	45581 (1.00)	99643 (1.00)
`147.vortex`	1351 (1.00)	29155 (1.00)	58907 (1.00)	137516 (1.00)
Total	7395 (1.00)	184670 (1.00)	366316 (1.00)	814584 (1.00)

(b) SPECint95 characteristics for *Alto/Base* binaries

Benchmark	Functions	Nodes	Edges	Instructions
`099.go`	602 (0.87)	14674 (0.90)	28825 (0.95)	74612 (0.92)
`124.m88ksim`	516 (0.85)	10317 (0.88)	20542 (0.92)	44959 (0.89)
`126.gcc`	2042 (0.91)	69092 (0.90)	146464 (0.93)	281908 (0.87)
`129.compress`	240 (0.72)	5002 (0.86)	9543 (0.90)	20607 (0.93)
`130.li`	602 (0.85)	8311 (0.84)	16485 (0.87)	32730 (0.83)
`132.ijpeg`	508 (0.68)	9459 (0.79)	18480 (0.83)	49486 (0.81)
`134.perl`	581 (0.81)	20190 (0.87)	41465 (0.91)	85195 (0.85)
`147.vortex`	962 (0.71)	24140 (0.83)	51966 (0.88)	108290 (0.79)
Total	6053 (0.82)	161185 (0.87)	333770 (0.91)	697787 (0.86)

(c) SPECint95 characteristics for *Alto/Inline* binaries

Benchmark	Functions	Nodes	Edges	Instructions
`099.go`	292 (0.42)	15511 (0.95)	30892 (1.01)	84105 (1.04)
`124.m88ksim`	361 (0.59)	12974 (1.11)	26097 (1.17)	60847 (1.20)
`126.gcc`	1492 (0.66)	73186 (0.95)	155106 (0.99)	307100 (0.95)
`129.compress`	162 (0.48)	6001 (1.03)	11367 (1.07)	25989 (1.17)
`130.li`	479 (0.67)	16721 (1.69)	33853 (1.78)	67290 (1.71)
`132.ijpeg`	359 (0.48)	9283 (0.77)	18369 (0.82)	49870 (0.82)
`134.perl`	412 (0.58)	25649 (1.11)	51607 (1.13)	111743 (1.12)
`147.vortex`	562 (0.41)	32030 (1.10)	69081 (1.17)	153009 (1.11)
Total	4119 (0.56)	191355 (1.04)	396372 (1.08)	859953 (1.06)

Table 3.4: Static characteristics of the SPEC95 integer benchmarks, for (a) *Original*, (b) *Alto/Base*, and (c) *Alto/Inline* binaries. Fraction relative to the original binaries are also presented in parenthesis.

3.3.1.2 Static characterization

Reporting measurements about the size of programs at different entity levels (e.g., instructions, basic blocks, etc.) not only provides information of the static characteristics of the programs under consideration, but also gives an estimation about the requirements needed for a compiler to process these programs. Table 3.4 summarizes most of the SPECint95 static characteristics. As the programs were statically compiled, the numbers include system libraries.

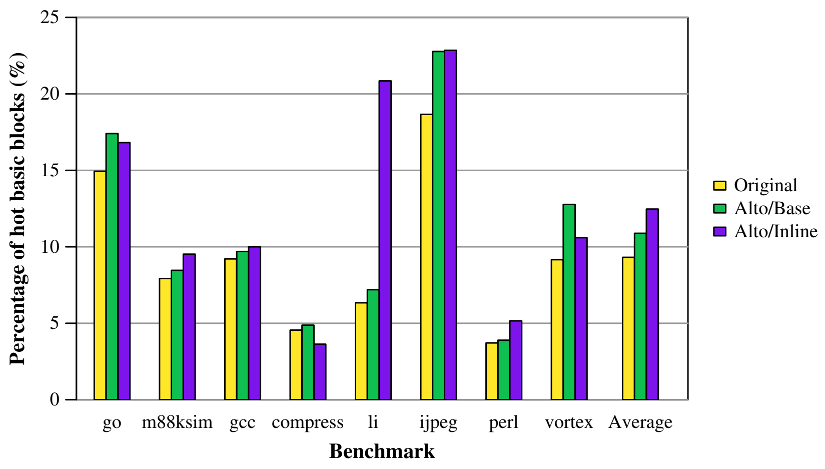


Figure 3.2: Percentage of *hot* basic blocks relative to the total number of basic blocks at compile time, for the SPECint95 programs. Profile information is used to decide whether a basic block is either *cold* or *hot*.

As we can see from the table, some benchmarks are relatively large. For example, program `gcc` has over 300K instructions, even for the *Original* benchmark set. These sizes are in general reduced when optimizations are applied, as we can see by looking at the relative fractions from Table 3.4b. Inlining, in contrast, increases the reported numbers in almost all programs, as shown in Table 3.4c. Hence, binary optimization algorithms must be aware of the potentially large size of the intermediate representation, causing algorithms that work well in conventional compilers to become not feasible because of their high time or space complexity. Furthermore, memory locality of the algorithms and data structures significantly influences performance at compile time.

Table 3.4 also shows some good news. As we can see from the *Alto/Base* binaries, optimizations performed by Alto seem to have a positive effect in program size, even though compaction is not the main goal of the Alto optimizer. As a result, procedure inlining reports slightly higher numbers on program size in most of the cases (see Table 3.4c), compared against our baseline. The most significant exception is, again, program `li`. As we can see, the number of static instructions increases about a 70%, which can explain the high relative compilation time we pointed out in Figure 3.1.

The above observation about program `li` is confirmed by looking at Figure 3.2, where we present the percentage of *hot* basic blocks relative to the total number of basic blocks in the program, when considering an execution threshold ϕ of 0.95. These results provide a good indirect estimation of Alto optimization time, since some of the optimizations are targeted at frequent portions of code. In general, the smaller the percentage of hot blocks, the lower the compilation time. As we can see, the percentage of hot basic blocks remains more or less

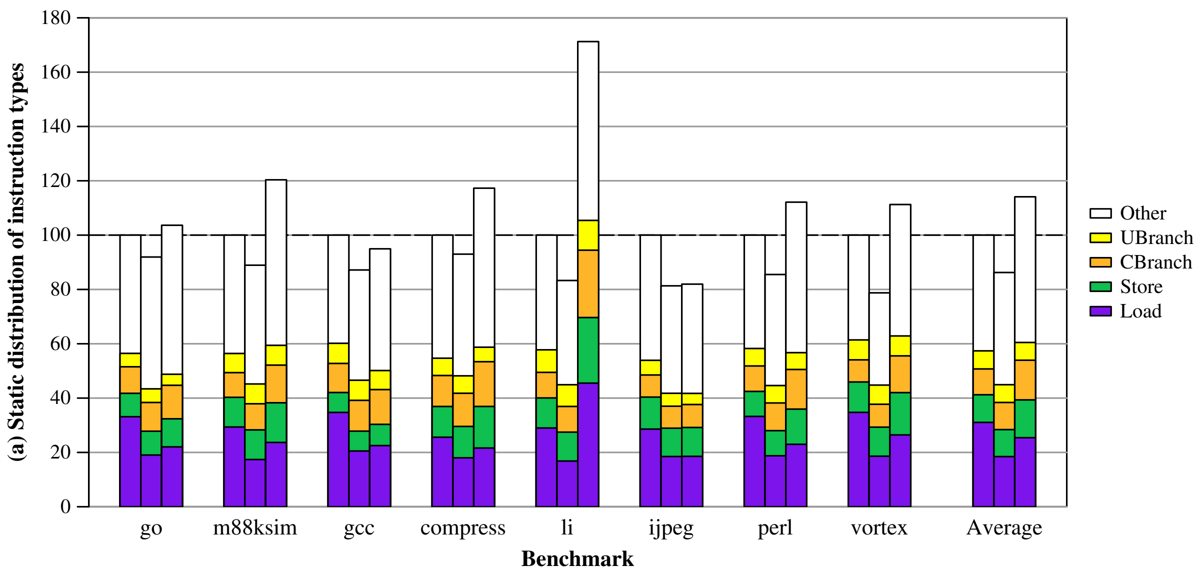


Figure 3.3: Static distribution of instruction types for the SPECint95 programs. Left, middle and right bars corresponds to *Original*, *Alto/Base*, and *Alto/Inline* executables, respectively.

the same after applying Alto optimizations, except for program `li` after applying procedure inlining. The resulting increment in this case (from 7% to 21% increase) is produced because inlined procedures are called from a significant number of call sites. Thus, when the callee procedure is inlined, both the original callee and the inlined copy still remain hot. This is the reason why Alto/Inline significantly increases compilation time for this program.

Finally, in Figure 3.3 we show the static distribution of instructions according to the instruction types. The important thing to note here is that conditional branches and memory references represent around 50% of static instructions in all programs, no matter what benchmark set is considered. This result points out that targeting binary redundancy elimination at these instructions may have greater benefits than do it for the rest of instruction types.

3.3.1.3 Dynamic characterization

We perform dynamic characterization of our SPECint95 benchmark sets by running all programs on top of the SimpleScalar `sim-profile` analyzer. This gives us a wide variety of statistics about the behavior of our programs at run time. For this process, we used the simulation inputs showed in Table 3.2.

The first dynamic measurements we present can be seen in Figure 3.4, which shows the dynamic distribution of instructions broken down by instruction type. The figure can be compared against the static results presented in Figure 3.3. Two points deserve to be mentioned about these results:

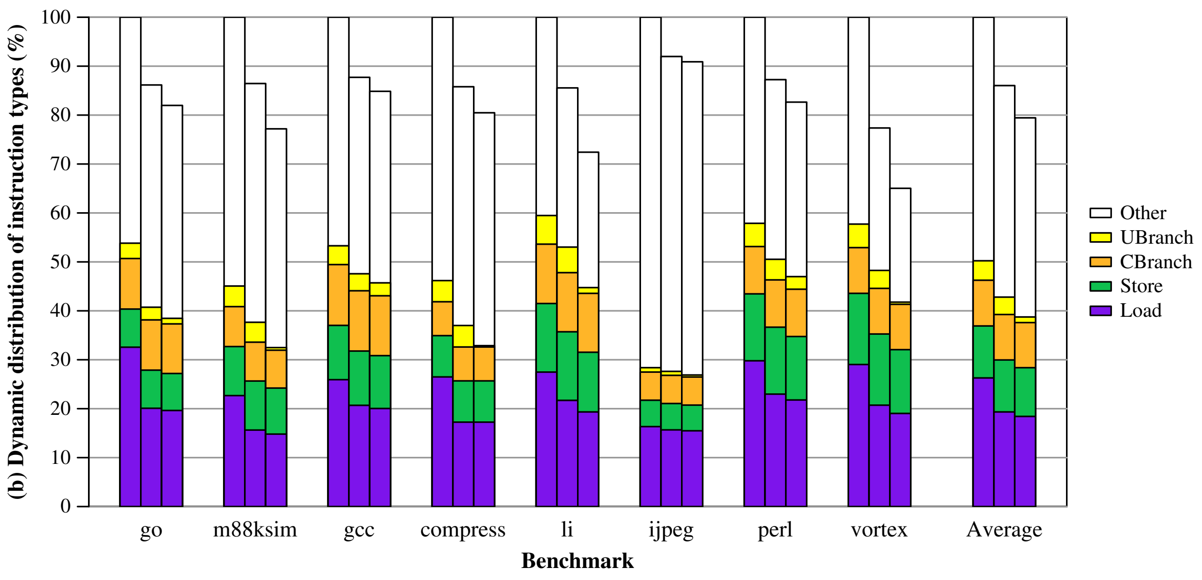


Figure 3.4: Dynamic distribution of instruction types for the SPECint95 programs, with simulation inputs from Table 3.2. Left, middle and right bars corresponds to *Original*, *Alto/Base*, and *Alto/Inline* executables, respectively.

- First, the 50% percentage of conditional branches and memory references is also valid for dynamic counts, which makes also valid our claim about targeting BRE to these instruction types.

- Second, and more important, binaries produced by Alto without inlining applied reduce the total number of executed instructions up to 14% in average, while when applying procedure inlining this percentage goes up to 20%. Furthermore, as we can see in the figure, reductions due to procedure inlining are more important on those programs where inlining was successful and aggressively applied, such as programs `m88ksim`, `vortex`, and specially, program `li`. The reason why we observe this effect is that procedure inlining allows to remove an important amount of dynamic call/return pairs (i.e., unconditional branches in our figure), as well as a significant percentage of load/store pairs that were originally inserted by the compiler to fulfill the OSF1 calling convention [Com99b]. As these load/store pairs have no sense when procedures are merged, Alto is allowed to remove them from the binary.

Collecting both static and dynamic statistics of the benchmark sets under consideration is certainly of interest to characterize the SPECint95 benchmark suite, but also to understand the effectiveness of the optimizations applied by Alto. However, the final measure of interest is to confirm whether actual execution time is reduced or not across the different benchmark sets.

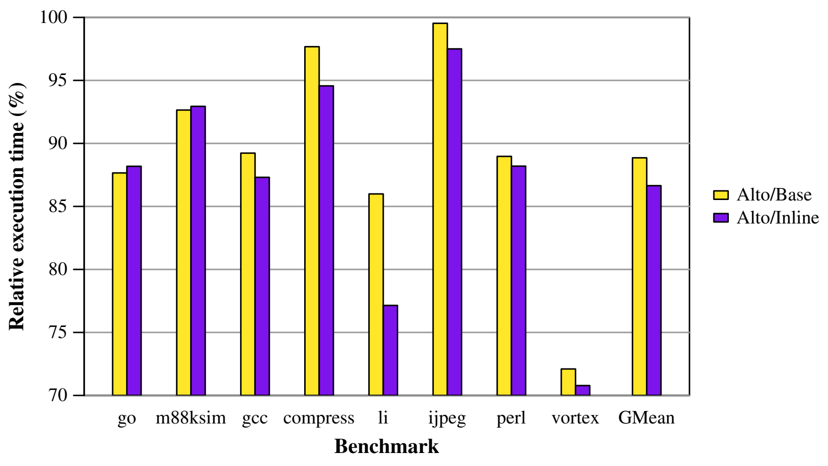


Figure 3.5: Effect of Alto optimizations in actual execution time, for the SPECint95 benchmarks using the input reference dataset. The baseline is *Original* binaries.

Figure 3.5 presents the relative execution time of the different programs and benchmark sets. These results were recorded by following the methodology that we mentioned in Section 3.1.3. In this case, as programs were not simulated but directly executed on the target environment, we used the official SPEC95 reference inputs (i.e., execution inputs, as shown in Table 3.2)

From the results presented in Figure 3.5 we can see that, for the majority of the programs, the executable optimized by Alto is considerably faster than the corresponding baseline. In several cases, the difference in the improvements is quite significant: for example, program `vortex` gets a 29% of improvement. Overall, the optimizations applied by Alto yield around 14% of execution time reduction. Note that these results are quite different in some cases to those reported by Muth [Mut99, MDWdB01]. However, neither the target platform nor the native compiler used are the same. Besides, we have improved some of the optimizations implemented by Alto, as we have explained in Section 3.2.1.2.

Finally, although the effect of inlining in execution time is in general beneficial, the observed mean is around 2% of reduction, with program `li` being the best case with a reduction of 9%. We believe that the main reason for this small improvement comes from the fact that implemented analysis within Alto are already intraprocedural, and that code layout is able to mitigate much of the locality effects of inlining. Besides, the observed slowdown in programs `go` and `m88ksim` indicate that some more fine tuning of inlining is probably necessary.

3.4 Methodology

Previous sections have presented the methodology we used for evaluating the benefits and drawbacks of the proposals developed in this work. In order to clarify this process, we next summarize in a list the steps we have followed:

1. All SPECint95 programs were compiled with full optimizations using the vendor-supplied C compiler on an AlphaServer GS–140 equipped with a 525MHz Alpha 21264 processor, on the platform presented in Section 3.1. For processing later by Alto, the compiler was also invoked with linker options to retain information and to produce statically linked executables (see Section 3.1.2.1).

2. The programs were then instrumented using Pixie [Smi91] and executed on the SPEC95 training inputs to obtain an execution frequency profile. Both basic block and edge counts will be later derived from this profile data, as shown in Section 3.2.1.2.

3. The programs and their corresponding profiles were then processed by Alto. The resulting executable programs are the baseline benchmarks used to evaluate the proposals presented in further chapters. That is, our baseline programs will be the fully-optimized benchmarks after being run through Alto.

4. As Alto will also be the compiler where our proposed algorithms will be implemented (see Section 3.2.1), a new set of benchmarks is then obtained from optimizing the original executable programs at the binary level, but this time including the particular optimizations in which we are interested.

5. Finally, programs will be either executed on the target platform for obtaining execution times, or simulated on top of our simulation tools, as Section 3.1.3 and Section 3.2.2 showed, respectively. If the latter is the case, different aspects of the programs will be measured. The results will be compared among the different benchmark sets obtained, starting from the baseline programs.

   The SPECint95 program inputs used for our simulation experiments were variants of the official SPEC input sets, to keep simulation time down to a manageable value (see Table 3.2) When programs were not simulated but directly executed on the target environment, the official "reference" inputs were used. In this case, the timings are obtained recording the smallest of seven runs of programs running in single-user mode.

By using the same methodology across all the work, we obtain consistent results that can be compared, which makes us feel confident about the validity of our experiments. Furthermore, our simulation results were validated by real machine runs.

Chapter 4

Alias analysis

In this chapter we review the problem of alias analysis *at the executable program level, identifying why memory disambiguation is one of the weak points of object code modification. Then, we propose several alias analyses to be applied in the context of link-time or executable code optimizers that are targeted to provide both* must- *and* may-*alias information. These analyses prove to be very useful for increasing memory disambiguation accuracy of binary code, which turns out into opportunities for eliminating binary redundancy.*

```
if (*p > 0)   //  Redundancy source
{
    *q = ...  //  Intervening store
}
if (*p == 1)  //  Redundant read
{
    ...
}
```

Figure 4.1: Example of binary redundancy from an alias analysis point of view.

4.1 Introduction

As we have seen in Section 2.2.3, code transformations on executable code can benefit greatly from pointer-alias information, as already happens with the compilation of source-level programs. For instance, whole program optimizations may open up opportunities for moving invariant memory instructions out of loops. However, alias information is key to identifying such instructions. Instruction scheduling and common subexpression elimination are other optimizations of limited usefulness in the absence of pointer-alias information.

As far as eliminating binary redundancy is concerned, having accurate alias information becomes a fundamental issue for detecting binary redundancies. For example, looking back to our old C code example presented now in Figure 4.1, in order to detect the existing memory redundancy two different alias information types are necessary:

***Must*-alias** First, we need to know if there is an *exact dependency* between the two `*p` references or not[1]. That is, both references *must* point the same memory location. In the example, this means that pointer `p` must not change between the two `*p` references.

***May*-alias** Also, we need to prove that there is no other memory write that *may* be in conflict with the memory location accessed by `*p`. That is, every other store must be *independent* with respect such memory location. In the example, this means checking whether pointer `q` *may* be aliased with pointer `p`.

Both types of alias information are in fact not only used when detecting binary redundancies, but also used for any optimization involving motion of memory references, where the relationship between pairs of memory instructions are needed. On the other hand, alias information will also be important for detecting other types of binary redundancy besides memory redundancy, as we will see in Chapter 6 for eliminating redundant conditional branches.

[1]Strictly, the dependency will not exist when both references are read accesses, such as in the example. However, we must consider such dependency so that order of instructions is preserved, since we do not want to change the redundancy direction.

While there is an extensive body of work on pointer alias analysis of various kinds [WL95, SH97, DWM98, CH00, GLS01], these are mostly high-level analyses carried out in terms of source language constructs. Unfortunately, such analyses turn out to be of limited utility at the machine code level. In fact, as we have seen in Section 2.2.3, the problem of memory disambiguation is one of the weak points of object code modification, because high-level information available in a traditional compiler is lost. Furthermore, features such as pointer arithmetic and out-of-bounds array accesses must be handled at this level, where the contents of every register is potentially an address.

In this chapter we present several alias analyses to be applied in the context of link-time or executable code optimizers. We can organize the proposed analyses in two groups:

1. First, we propose a new high-accurate *must*-alias analysis to recognize memory dependencies in a *path-sensitive* fashion. The analysis is based on the idea of establishing alias relationships for only a subset of all the possible paths between every pair of references to disambiguate. Thus, this alias information is particularly well suited to be used for eliminating path-sensitive redundant memory operations, as we will see in Chapter 5.

2. Next, we also propose two approaches to high-quality, low-cost, speculative *may*-alias analysis to recognize memory independencies. The key idea behind these proposals is to trade off analysis complexity against *safeness*. Our alias analysis incorporate in their data-flow equations the notion of "guessing" when two memory references are *most likely* independent. By being more liberal in the propagation and use of the data-flow information, we increase alias precision on important portions of code while keeping the analysis reasonably cost-efficient, yet the analyses may sometimes yield wrong answers[2].

The alias algorithms we propose are targeted to provide either *must*- and *may*-alias information, respectively. Our results show that these algorithms prove to be very useful for increasing memory disambiguation accuracy of binary code, which turns out into opportunities for applying optimizations such as eliminating binary redundancy.

4.2 Path-sensitive *must*-alias analysis

The problem of alias analysis or *memory disambiguation* at the machine code level is to determine the relationship of every pair of memory references in a program. A common approach in compile-time instruction schedulers is called *disambiguation by instruction inspection*, which is a global scheme based on *register use-def chains* (see Section 2.2.3.1 and Section 2.2.2, respectively). Register use-def chains provide, for each use of a register, a pointer to its definition.

[2]Of course, applying speculative optimizations on top of these speculative analyses will require recovery code to compensate in those cases where the memory disambiguator was wrong.

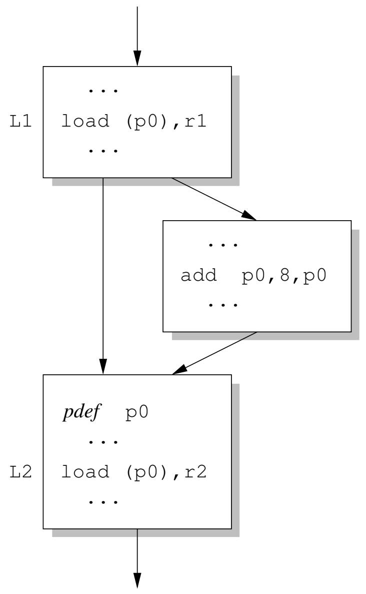


Figure 4.2: Example of memory references where general inspection fails for disambiguation. Memory disambiguation must be performed in a path-sensitive fashion.

Although this strategy is able to disambiguate a significant percentage of memory references, it fails in the general case. Figure 4.2 shows an example where instruction inspection is not able to expose the relationship between load instructions L_1 and L_2. The reason is that, as several definitions for pointer p_0 reach the use in load L_2, the two loads handle potentially different definitions for their base registers. As a result, the alias analysis we use for disambiguating memory references, like most analyses used in optimizing compilers, is unable to disambiguate references in a path-sensitive way.

The problem with *path-sensitive disambiguation*, unfortunately, is that the compiler has to pay the *exponential* price of analyzing each path separately [Bod99]. The reason why analyzers avoid this situation is that, even in a program with no loops, there is an exponential number of paths. To stay practical, analyzers treat paths together, summarizing their results whenever paths meet, therefore diluting optimization opportunities.

In this section, we present a new technique for detecting exact memory dependencies in a path-sensitive fashion, that is, to recognize *path-sensitive memory dependencies*. The key to our new proposal is to extend the formulation of the general memory disambiguation algorithm to effectively analyze each path separately. Furthermore, we apply simple but effective heuristics to reduce the exponential cost of the algorithm and keep it under control. The resulting alias information can then be used to guide compiler optimizations based on detecting path-sensitive dependencies, as we will show in Chapter 5.

4.2.1 Alias analysis by instruction inspection

Memory disambiguation at the machine code level is usually based on a global scheme named *register use-def chains*, which provides, for each use of a machine register, a pointer to its definition (see Section 2.2.2). The use-def chains are a directed graph whose nodes are instructions and whose edges are use-def pointers.

When there are several definitions of a register reaching a use, it is common to introduce a pseudo instruction at an appropriate place which also defines that register, thereby shadowing the other definitions. This is analogous to ϕ functions used with the static single assignment (SSA) form [Muc97f]. Therefore, for each instruction in the flow graph, the algorithm derives a symbolic description for every one of its source registers.

Definition 4.2.1 *Let r be a source register for an instruction I. A* symbolic descriptor *$\mathcal{S}$ for register r and instruction I is a pair $\langle\alpha, c\rangle$, where α is either an instruction or a pseudo-instruction, and c is an integer value. Given a symbolic descriptor $\mathcal{S} = \langle\alpha, c\rangle$, the instruction α is said to be the* defining instruction *for register r, while c is called the* offset *relative to the value computed by instruction α.* $\square$

The above definition is then used for defining a new kind of descriptor for memory instructions, which will be used for disambiguation.

Definition 4.2.2 *Let M be a memory instruction accessing the location whose address is $(r) + k$, where r and (r) are the instruction* base register *and its content, respectively; and k is an integer constant. A* memory descriptor *$\mathcal{M}$ is a pair $\langle\mathcal{S}, k\rangle$, where $\mathcal{S}$ is the symbolic descriptor for register r and memory instruction M.* $\square$

An *instruction inspection* algorithm compares memory descriptors for checking their relationship (see Section 2.2.3.1). This disambiguation approach is able to deal with register copies and address arithmetic across basic block boundaries. However, disambiguation is only allowed under certain conditions.

Definition 4.2.3 *Let $\mathcal{M}_1 = \langle\langle\alpha_1, c_1\rangle, k_1\rangle$ and $\mathcal{M}_2 = \langle\langle\alpha_2, c_2\rangle, k_2\rangle$ be two memory descriptors for memory instructions M_1 and M_2, respectively. Then, $\mathcal{M}_2$ is* disambiguable *with respect to $\mathcal{M}_1$, written $\mathcal{M}_1 \prec \mathcal{M}_2$,* **iff** *(i) M_1 and M_2 are known to access different memory regions;* **or** *(ii) $\alpha_1 = \alpha_2$ and there is an existing path leading from α_1 to M_2 where M_1 is reached.* $\square$

From the Definition 4.2.3, at least one of the following two conditions must succeed for allowing safe disambiguation. Condition *(i)* is referring to instructions that are known to access different memory regions, such as stack and global sections. Condition *(ii)* is a bit more complex:

1. Both descriptors must share the defining instruction, and

2. Instruction M_1 must be "located before" instruction M_2 in the flow graph.

Alias relationship is then established by checking the relation $c_1 + k_1 = c_2 + k_2$. Otherwise, instructions cannot be disambiguated and they are assumed to have an unknown dependency. Note that $\prec$, unlike traditional alias disambiguators, is neither reflexive nor symmetric. Besides, its independency on dominator and loop information makes it valuable for executable code, where this information is not always available.

4.2.2 Path-sensitive memory disambiguation

When general disambiguation is not possible, our instruction inspection algorithm will check for *path-sensitive disambiguation*. That is, alias relationships is established for only a subset of all the possible paths between the pair of references to disambiguate. To this end, we first formulate the following definition:

Definition 4.2.4 *Let* $\mathcal{M}_1 = \langle\langle \alpha_1, c_1\rangle, k_1\rangle$ *and* $\mathcal{M}_2 = \langle\langle \mathbf{a}_2, c_2\rangle, k_2\rangle$ *be two memory descriptors for memory instructions* M_1 *and* M_2 *respectively, where* $\mathbf{a}_2 = \phi(\langle \alpha_{2,1}, c_{2,1}\rangle, \ldots, \langle \alpha_{2,n}, c_{2,n}\rangle)$*, and* $n > 1$*. Then, a* chain of symbolic descriptors $\omega = \langle \beta_1, d_1\rangle, \ldots, \langle \beta_m, d_m\rangle$*, for memory descriptors* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, is a sequence of descriptors such that (i)* $m > 1$ *and* $\forall_{i<m}\ \beta_i$ *is a pseudo-instruction, (ii)* $\beta_1 = \mathbf{a}_2$*,* $d_1 = c_2$ *and* $\beta_m = \alpha_1$*, and (iii)* $\forall_{i,j}\ \beta_i = \beta_j \iff i = j$*.* $\square$

The chain of symbolic descriptors ω is simply a back-sequence of use-def chains starting from M_2 to α_1, where every but the last item are symbolic descriptors containing unique pseudo-instructions. The property that we want to check is then as follows:

Definition 4.2.5 *Let* $\mathcal{M}_1 = \langle\langle \alpha_1, c_1\rangle, k_1\rangle$ *and* $\mathcal{M}_2 = \langle\langle \mathbf{a}_2, c_2\rangle, k_2\rangle$ *be two memory descriptors for memory instructions* M_1 *and* M_2 *respectively. Then,* $\mathcal{M}_2$ *is* path-sensitive disambiguable *with respect to* $\mathcal{M}_1$*, written* $\mathcal{M}_1 \prec_{\text{ps}} \mathcal{M}_2$*,* **iff** *(i)* $\mathcal{M}_1 \prec \mathcal{M}_2$*;* **or** *(ii) there is an existing chain of symbolic descriptors* ω*, for memory descriptors* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, exposing a path leading from* α_1 *to* M_2 *where* M_1 *is reached.* $\square$

It is straightforward to understand that path-sensitive disambiguation defined above subsumes generic disambiguation presented in Definition 4.2.3. If a ω chain exists, disambiguation is then performed by checking the relation $c_1 + k_1 = d_1 + \ldots + d_m + k_2$. The result of this check will determine the alias relationship between the considered pair of instructions, which will be dependent on the path defined by all the possible ω's for which the definition holds. The next example will help understanding how the algorithm works.

Example 4.2.1 *From Figure 4.2, let I_0 be the instruction defining register p_0, assuming that I_0 is placed before L_1 in the same basic block; and let I_3 be the* `add` *instruction. From Definition 4.2.1 and Definition 4.2.2 we derive memory descriptors $\mathcal{M}_1 = \langle\langle I_0, 0\rangle, 0\rangle$ for reference L_1, and $\mathcal{M}_2 = \langle\langle \mathbf{a}_2, 0\rangle, 0\rangle$ for reference L_2, being $\mathbf{a}_2 = \phi(\langle I_0, 0\rangle, \langle I_0, 8\rangle)$. Note that symbolic descriptor $\langle I_0, 8\rangle$ results from combining* partial *descriptors $\langle I_3, 0\rangle$ and $\langle I_0, 8\rangle$. Then, for disambiguating L_1 and L_2 references, the algorithm performs the following steps:*

1. *First, the algorithm checks for $\mathcal{M}_1 \prec \mathcal{M}_2$ (i.e., Definition 4.2.3). However, disambiguation is not allowed because $I_0 \neq \mathbf{a}_2$. That is, they do not share the same defining instruction.*

2. *The algorithm checks then for path-sensitive disambiguation. It is straightforward to see that $\mathcal{M}_1 \prec_{\text{ps}} \mathcal{M}_2$, since $\omega_1 = \langle \mathbf{a}_2, 0\rangle, \langle I_0, 0\rangle$ and $\omega_2 = \langle \mathbf{a}_2, 0\rangle, \langle I_0, 8\rangle$ expose the desired property on the left and right paths, respectively.*

As a result, ω_1 exposes a dependency relationship on the left path, because the result of accumulating $\mathcal{M}_2$'s and ω_1's offsets is equal to $\mathcal{M}_1$'s offset (i.e., zero in our example). This is not true for ω_2, which exposes a memory independency on the right path. As we will see in Chapter 5, we can conclude from the above results that L_2 is path-sensitive redundant *with respect to L_1, but only on its left path.* □

Due to the exponential price of dealing with each path separately, the scheme for path-sensitive disambiguation does not come without a cost, since establishing a path-sensitive relationship is a *bidirectional problem*. As widely known [Muc97f], bidirectional problems require forward and backward propagation at the same time and are significantly more complicated to formulate, understand, and solve than one-directional problems. Since our current implementation uses a backtracking algorithm, we reduce the high computational cost of the algorithm by using a couple of simple but effective heuristics when checking for ω:

1. For every i-item of ω, where $1 < i < m$, we consider only symbolic descriptors that belong to a *hot path* of the program.

2. We do not allow $m > K$, where K is a fixed constant[3].

We have observed that by using these heuristics the algorithm misses only a few opportunities for disambiguation, while compilation time does not significantly increase (as we will show in Section 5.7.4).

[3]In our implementation we have used a K value of 5.

Method	**Description**
Inspection	Corresponds to the disambiguation mechanism by instruction inspection, using path-insensitive use-def chains (Section 2.2.3.1).
Inspection$_{PS}$	Corresponds to the disambiguation mechanism by instruction inspection, using path-sensitive use-def chains (Section 4.2.2).

Table 4.1: Description of must-alias analysis methods for memory disambiguation.

4.2.3 Evaluation

In this section, we describe the process we have followed for evaluating the effectiveness of the proposed path-sensitive must-alias analysis.

We have implemented the must-alias analysis algorithms presented in Table 4.1 (and described in the previous sections) on the Alto framework we described in Section 3.2.1. The information reported here was then obtained after several optimization rounds carried out by Alto, such as constant/copy propagation, dead/unreachable code elimination, inlining, etc. To compute must-alias information for later disambiguation, an interprocedural data-flow analysis computes the *use-def* chains (see Section 2.2.3.1). This is the only information required for detecting both path-insensitive and path-sensitive must-alias information, as it has been shown in Section 4.2. For our experiments we have used an execution threshold ϕ of 0.6 (see Section 3.2.1.3).

Memory disambiguation for a particular pair of memory references is applied incrementally, following the scheme for must-alias disambiguation presented in Figure 4.3. We can observe that when general instruction inspection fails, disambiguation is made by checking use-def chains in a path-sensitive fashion, as we have shown in Section 4.2.2. Note also that, as more than one single relationship due to different paths can be obtained in this case, we are only interested in whether such relationship could be established or not.

The benchmarks we have used for our experiments were presented in Section 3.3, and they were generated following the methodology described in Section 3.4. For our experiments, we have chosen the *Alto/Inline* binaries as a baseline[4] (that is, the highest optimized binaries we have), whose characteristics and generation procedure were described in Section 3.3.1.

4.2.3.1 Measuring static precision

We start evaluating the effectiveness of each disambiguation mechanism described in Table 4.1 by comparing their static accuracy in terms of "disambiguation queries". A *disambiguation query* is a question made to the memory disambiguator about the relationship between two

[4]We have not considered the effect of disabling inlining in this chapter. The reasons are that (a) we will already observe these effects on the path-sensitive disambiguation when evaluating our proposals in Chapter 5, and (b) as our may-alias analyses are interprocedural algorithms, they will not be very sensitive to inlining.

```
Input: Two memory instructions $I_1, I_2$.
Output: An alias relationship {dependent, independent, path-sensitive relationship, unknown}.
Method:
    if ud-chains($I_1, I_2$,path-insensitive) $\neq$ unknown then
        return ud-chains($I_1, I_2$,path-insensitive);
    elsif ud-chains($I_1, I_2$,path-sensitive) $\neq$ unknown then
        return path-sensitive relationship;
    else
        return unknown;
    endif
End Method
```

Figure 4.3: Path-sensitive memory disambiguation scheme, for a pair of memory instructions.

memory instructions. Thus, the returned value for the path-sensitive disambiguator defined in Figure 4.3 can be *dependent*, *independent*, *path-sensitive relationship* or *unknown*. We are not really interested in discovering the exact type of relationship (i.e., dependency or independency), but rather in whether the analysis returned an answer different than "unknown". We consider that a given alias analysis "is better" if it returns less "unknown" responses.

Since our different alias analyses are not being driven by any particular optimization, we have generated a representative set of queries by using the following algorithm. First, we consider every load/store instruction within the *hot basic blocks* of every function[5]. Then, for every candidate, we start looking back over all (both *cold* and *hot*) paths for load/store instructions that reach the candidate. For each load/store instruction found and its candidate, a query is made to the corresponding disambiguator. We believe this scheme faithfully mimics the typical behavior of many compiler optimizations, which will only generate queries about pairs of instructions that are connected by an existing path.

For evaluating the precision of our path-sensitive must-alias disambiguation method (i.e., $Inspection_{PS}$, see Table 4.1) we applied the disambiguation scheme presented in Figure 4.3 to our set of queries. The result is presented in Figure 4.4, which shows the relative percentage of queries successfully resolved by the $Inspection_{PS}$ method[6]. The baseline is in this case the percentage of queries resolved by the pure *Inspection* scheme, that is, without considering path-sensitive information at all.

From the results presented in Figure 4.4, we can see that our proposed scheme for path-sensitive disambiguation is able to increase must-alias precision up to 150% in geometric mean with respect to the path-insensitive baseline. In particular, precision increases for almost every benchmark in more than 25%, with some better cases of around $2x$ for programs `go`

[5]We choose instructions only from the *hot* path because we are only interested in measuring the precision of critical instructions in a program.

[6]A particular query is *successfully resolved* if its result is different than "unknown".

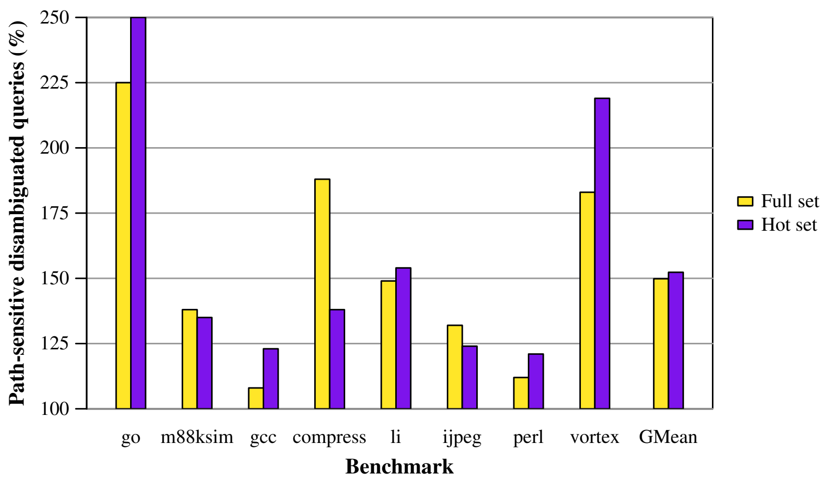


Figure 4.4: Precision of the path-sensitive must-alias memory disambiguation scheme, with respect to the path-insensitive must-alias disambiguation method as a baseline (i.e., *Inspection*$_{PS}$ vs. *Inspection*). The left bar considers the full set of queries, while the right bar restricts the set to those queries where both components are instructions from *hot* paths.

and `vortex`. As we will see in Chapter 5, this rise is significant enough for considering the detection and elimination of further path-sensitive memory redundancies.

4.3 Speculative *may*-alias analysis

Typically, complexity of data-flow analyses has been a compromise between *cost* and *precision* (see, for example, [ASU86a, Muc97c, BA98, DGS97, RRL99]). Thus, the higher the precision desired, the harder to keep the algorithm space and time feasible. For high-level compilation, compiler writers have tended to use sophisticated analysis at the expense of increased resource usage. However, given that statically-linked executable programs tend to be significantly larger than the corresponding source level entities, traditional analyses applied to machine code level are of limited usefulness, because either the cost is too high or the precision is not accurate enough.

The key idea behind this proposal is to introduce a new variable in the game: *safeness*. Breaking the strong constraint of safeness, a data-flow analysis may reach a high level of precision at low cost, by paying the price of not always being correct. In other words, the data-flow analysis becomes *speculative*, or *unsafe*. As far as we know, this is the first attempt to systematically introduce unsafe speculations into data-flow analysis algorithms.

In the following sections we introduce two interprocedural data-flow algorithms that increase *may*-alias analysis accuracy of binary code by using *speculation*. The two algorithms are orthogonal and, thus, can be applied independently or coupled together.

- The first algorithm tries to disambiguate memory references by classifying them into separate memory regions (e.g., heap, stack and global) and unsafely assumes that whenever an arithmetic operation is performed on a pointer, the pointer will not change its pointed-to memory region.
- The second algorithm uses profile information and unsafely assumes that memory instructions on hot paths are not aliased to memory references on cold paths.

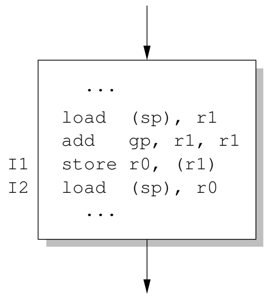
By making these speculative assumptions, we obtain more precise information in the common case, yet the analysis results are not always correct. This means that any optimization performed by using these speculative analyses will be speculative as well, and some type of check-and-recovery mechanism must be provided. We will extend this discussion in Section 4.3.3, although speculative optimizations and recovery mechanisms are beyond the scope of this work.

4.3.1 Region-based speculative alias analysis

In Section 2.2.3.2 we presented an interprocedural algorithm proposed by Debray *et al.* [DMW98] for reasoning about may-alias information of executable code, which is able to to expose memory independencies. *Residue-based alias analysis* overcomes some of the limitations of disambiguating references by instruction inspection. (see Section 2.2.3 for an overview of the most common techniques). However, this algorithm fails in several situations, leading to an undesirable loss in precision:

- First, by using mod-k residues, the algorithm is clearly oriented to “fine grain” disambiguation, but it is unable to effectively catch “coarse grain” alias relationships (e.g., whether two references point to different memory regions).
- Second, in order to keep the algorithm space and time feasible, they apply a conservative widening operation that causes information to be lost when joining definitions of the same register from different control-flow paths. This is specially negative for pointer arguments at the entry node of functions with multiple call sites, since the context-insensitive nature of the algorithm leads to a massive application of the widening operation.
- Furthermore, the algorithm does not keep track of memory contents, which causes information to be lost when registers are saved/restored.

Figure 4.5 shows a situation where the residue-based analysis presented in Section 2.2.3.2 is not accurate enough. As it can be seen, the value of register r_1 is defined by a load instruction. Therefore, the analysis in unable to propagate information through the uses of



```
       ...
     load  (sp), r1
     add   gp, r1, r1
I1   store r0, (r1)
I2   load  (sp), r0
       ...
```

Figure 4.5: Sample code where pointer information is lost because a component of the *use-def* chain of r_1 is defined by a load operation.

r_1. Yet, it is very *unlikely* that register r_1 at instruction I_1 points to regions other than the global data area because the value loaded from memory was operated previously with the pointer-to-global gp register. Thus, the safeness of the residue-based alias analysis is missing a very likely opportunity of disambiguating the two memory references[7]. To overcome this drawback, our first proposal is to propagate which *memory regions* a register may point to, instead of being worried about symbolic descriptors based on instructions which may define that register.

Definition 4.3.1 *A* region descriptor α_r^p *for a register r at program point p is a subset of the finite set of values $\{\mathcal{G}, \mathcal{S}, \mathcal{H}\}$, denoting all possible memory regions (i.e.,* global*,* stack *and* heap*, respectively) pointed by register r at program point p.* $\square$

For a particular region descriptor, a value of $\emptyset$ denotes that register is not used as a pointer to any memory region, and is written as $\top$; while a value of $\{\mathcal{G}, \mathcal{S}, \mathcal{H}\}$ denotes a total lack of information, and is written as $\perp$. This information will be then propagated using a general data-flow iterative algorithm, over the lattice presented in Figure 4.6. The input values, for every register r at every program point p, are initialized as follows:

$$\alpha_r^p = \begin{cases} \{\mathcal{G}\} & \text{if } r = gp \text{ (global pointer)} \\ \{\mathcal{S}\} & \text{if } r = sp \text{ (stack pointer)} \\ \top & \text{otherwise} \end{cases}$$

The value $\{\mathcal{H}\}$, which denotes a pointer to the *heap* memory area, is assigned to the destination register of the system call *break*, at such program point. System call *break* is used by Unix-based operating systems for heap management, through the allocation/deallocation library functions `malloc` and `free`.

[7]Note that this loss of precision would also happen if the descriptor of r_1 had been mapped to $\perp$ due to the application of a widening operation at the basic block entry.

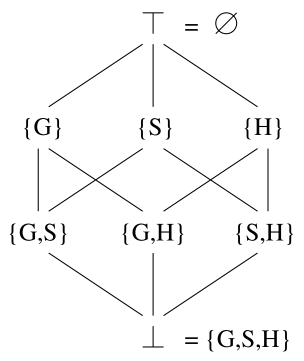


Figure 4.6: Region-based alias analysis lattice, representing the possible set of regions that can be assigned to a region descriptor.

When propagating information through the control-flow graph, the effect of instructions on its corresponding destination register may vary. For example, a load instruction sets the descriptor of its destination register to the value $\perp$, since no information about the contents of memory cells is kept in the algorithm. Other additional instructions, such as conditional and unconditional jumps, have the only effect of determining the control-flow graph of the program. Therefore, they are not considered explicitly in the context of our alias analysis. We also ignore operations on floating point registers, assuming that such operations will not be used for address computation. For the rest of instructions, the general behavior is as follows:

Definition 4.3.2 *Let $op^p\ r_i, r_j, r_k$ be an instruction at program point p, with source registers r_i, r_j and destination register r_k. Let α_i^p and β_j^p be the region descriptors for registers r_i and r_j, respectively. Then, the region descriptor γ_k^{p+1} for register r_k at program point $p+1$, is set to $\alpha_i^p \bullet \beta_j^p$, defining the $\bullet$ operator as:*

$$\alpha_i^p \bullet \beta_j^p = \begin{cases} \alpha_i^p & \text{if } \alpha_i^p \neq \top \wedge \beta_j^p = \perp \\ \beta_j^p & \text{if } \beta_j^p \neq \top \wedge \alpha_i^p = \perp \\ \alpha_i^p \cup \beta_j^p & \text{otherwise} \end{cases}$$

□

In the case that a source operand is a constant instead of a register, the corresponding region descriptor is assumed to be $\top$. Note that a region descriptor with a value different than $\top$ being operated with a value of $\perp$ will propagate the non-$\top$ descriptor to the instruction destination register. Strictly, it is *unsafe* to make such an assumption, although the opposite rarely occurs. Furthermore, propagating the non-$\top$ description in this way may carry some problems out in terms of correctness of the analysis. We will discuss later these issues, in Section 4.3.1.2.

If a node of the control-flow graph has more than one predecessor, the information stemming from these predecessors must be integrated. In data-flow frameworks, joining paths in the flow graph is implemented by the union operator.

Definition 4.3.3 *Let α_r^p and β_r^p be region descriptors for register r at program point p, coming from two different predecessors. Then, the join operation $\bigtriangledown$ is defined as:*

$$\alpha_r^p \bigtriangledown \beta_r^p = \begin{cases} \bot & \text{if } \alpha_r^p = \bot \vee \beta_r^p = \bot \\ \alpha_r^p \cup \beta_r^p & \text{otherwise} \end{cases}$$

□

Note that region-based join operation differs from residue-based approach in that "widening" does not need to be applied. In this case, a simple union between region descriptors is performed, which avoids loosing all the information at that program point. Therefore, join of paths in this case is not so conservative.

The *region-based alias analysis* presented here is complementary to any *points-to* alias scheme, and may be coupled to it, or may be computed separately. As far as the analysis implementation is concerned, not only the three mentioned regions need to be considered (i.e., global, stack and heap), but whatever other set of N memory regions, such as uninitialized/initialized global data, different stack frames, etc. The only constraint is that considered regions need to be *disjoint*. A region descriptor may be then represented as a K-bit vector, where every bit denotes one of the considered memory regions. The resulting analysis requires only KRN bits of memory for a program with N basic blocks on a machine with R registers.

4.3.1.1 Dealing with memory contents

Much of the loss in precision of the residue and region-based analysis approaches comes from the fact that both algorithms do not keep track of the contents of memory when registers are saved/restored. For example, looking back to Figure 4.5, the region-based alias analysis would set the region descriptor of register r_1 to $\bot$, loosing all the information kept in the source register of a possible previous store instruction to the same location. The obvious solution would be to propagate values (i.e., residue-based, region-based, or whatever other symbolic descriptors) through memory cells. However, such an analysis implies a high increase in memory requirements.

Since our proposal is to reduce cost in exchange for increased precision, even when this may carry unsafe results, a cheap alternative would be to assume that result of a load operation will not point to memory regions. That is, its region descriptor will be mapped to $\top$. This is true for a high number of loads in a program, since values loaded from memory are frequently not used as memory pointers. However, such assumption may fall into increasingly unsafe results. Therefore, this heuristic should be used carefully.

4.3.1.2 Reasoning about data-flow analysis correctness

When propagating region descriptors through instructions in Definition 4.3.2, we noted that non-$\top$ region descriptors are propagated in those situations where a value of $\perp$ appears as the other region descriptor to be operated. This operation is the very essence of the speculative behavior of the region-based alias analysis. However, such behavior expose a couple of issues that need to be mentioned:

1. Strictly, it is *unsafe* to make such an assumption, although the opposite rarely occurs. For instance, a C code sequence might produce from a pointer to the global data area, a pointer to the program stack, but it is uncommon for many programs to generate these types of accesses. We did not found such scenarios in our benchmark suite, but operating system kernels and tools, as well as virtual machines are programs where this situation might happen. Certainly, our proposed analysis could be run as a safe analysis on user-demand by some compiler command line option, like actual production compilers do on several unsafe optimizations.

2. Propagating the non-$\top$ descriptor as described yields an incorrect lattice because the $\perp$ element does not *attract* the rest of the possible region descriptors. Hence, the iterative solving algorithms might not converge in all situations [Muc97f, NNH99]. To avoid this non-convergence to happen, we only allow a maximum number C of iterations, where C is a fixed constant[8]. As a result, the algorithm might produce incorrect results for some region descriptors, although this inaccuracy is somehow less important since the analysis is already unsafe by itself.

   In our benchmark suite, however, we did not found such scenario either. Actually, most of the times the algorithm is able to converge in a few iterations. We believe this is because, in general, it is very uncommon to reach a $\perp$ value by simply joining the different regions from incoming predecessors at the entry of nodes.

4.3.2 Profile-guided speculative alias analysis

When trying to keep the residue-based alias analysis algorithm space and time feasible, the conservative widening operation does not join definitions of the same register from different control-flow paths in a set. For this reason, when computing the meet of the incoming information at the entry of the basic block, the information associated with the register is widened to $\perp$. That is, all the information is lost. An example of this can be seen in Figure 4.7, where register r_1 is defined from two different instructions. In the example, region-based alias analysis does not solve the problem either, since joining regions also fails in this case. This turns out to lead to an undesirable loss in precision in a number of situations.

[8]In our implementation we have used a C value of 6.

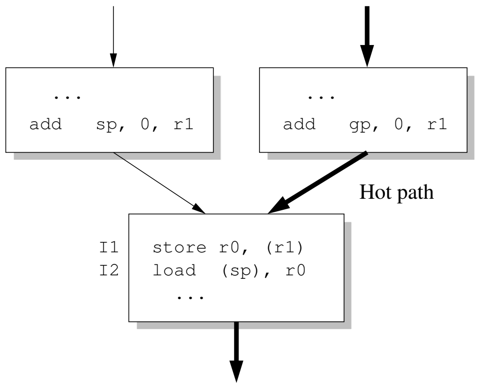


Figure 4.7: Sample code where different definitions are reaching a use, but there is a more likely executed path.

The preceding situation also occurs on pointer arguments at the entry node of functions, due to the context-insensitive formulation of the analysis. A possible solution would be to use a context-sensitive interprocedural approach. However, the defining instructions for a register are generally different at different call sites to a function, which means that the callee will have to be analyzed separately for each such call site. Given that statically-linked executable programs tend to be significantly larger than the corresponding source level entities, this indicates that the cost of a traditional context-sensitive analysis is likely to be quite high.

For our second speculative alias analysis proposal we choose, instead, to use a profile-guided analysis. The basic idea is to propagate alias information only for important paths, ignoring those paths whose information will cause loss of precision in the most common cases. As a result, the application of inaccurate join operations to $\perp$ will be drastically reduced. For instance, looking back to Figure 4.7, we can see that the most likely definition of register r_1 is the one from register gp. We could then easily determine by a single inspection that accesses at instructions I_1 and I_2 are likely to be disjoint. More formally:

Definition 4.3.4 *Let $\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}$ and $\beta_r^{p,1}, \ldots, \beta_r^{p,m}$ be the set of symbolic descriptors for register r at program point p, coming from predecessors $1, \ldots, n$ and $1, \ldots, m$ connected to p by hot and cold edges, respectively; and let $\bigtriangledown$ be a join operator, the new join operation $\bigtriangledown_{spec}$ is defined as:*

$$\bigtriangledown_{spec}\left(\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}, \beta_r^{p,1}, \ldots, \beta_r^{p,m}\right) = \bigtriangledown\left(\alpha_r^{p,1}, \ldots, \alpha_r^{p,n}\right)$$

□

That is, the algorithm only takes into account the information coming from *hot* edges of the control-flow graph. Note that the result of this new meet operation is *speculative* in nature, because we simply "ignore" some possible (although infrequent) paths in the analysis.

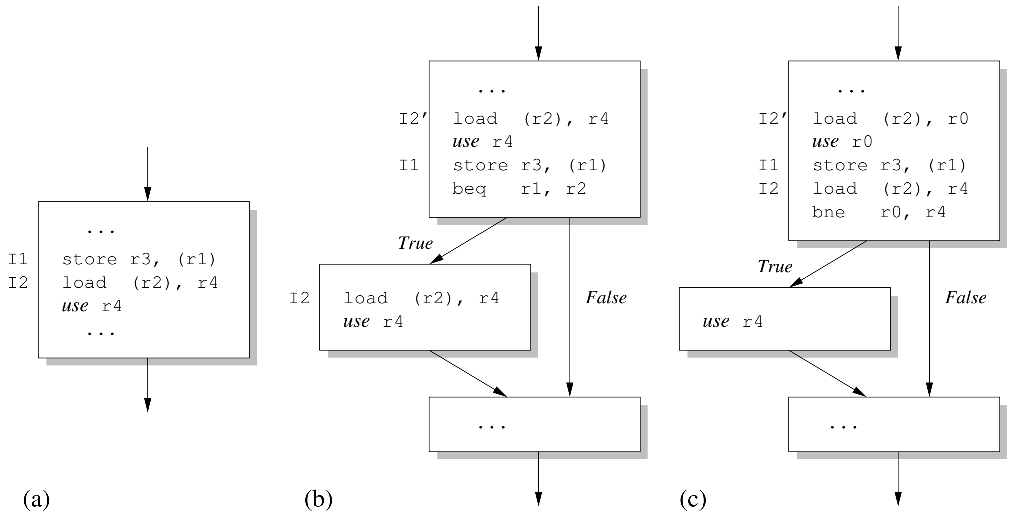


Figure 4.8: Reordering memory operations [MM97]: (a) original sample sequence; recovery-based reordering by using (b) *interference test*, (c) *coherence test*.

This will give more precise information in the common case, but the result of the analysis will not always be correct.

This simple speculative data-flow scheme is general enough to be applied on top of any traditional data-flow analysis algorithm, not necessarily related to neither pointer aliasing nor machine code level. In particular, this proposal is orthogonal to the one described in Section 4.3.1. On the other hand, the cost of a speculative data-flow technique does not change with respect to the non-speculative safe version which it is based on. However, the intuition indicates that by using profile information to avoid analyzing unimportant paths, significant reductions in the space and time requirements for the analysis can be achieved.

4.3.3 Recovery-based usage of speculative alias analysis

The proposals presented in this section increase the precision of the may-alias analysis by providing more reliable information in the common case, at low cost. However, the speculative nature of our schemes causes the analysis results to be not always correct. This means that any optimization performed using this speculative analysis will be speculative as well.

Speculative optimizations [HSS94, GCM$^+$94, MM97, GBF98, PGM00a] have been widely used in the compiler world for reducing the overall execution time of programs. The key idea behind speculation is breaking the original program sequence by executing a (possibly *unsafe*) "better" reordering of instructions, corresponding to the most likely execution paths. Since the new executed sequence may be unsafe, some type of *check-and-recovery* mechanism must be provided for validating/undoing such assumptions at run time. In this mechanism, "checking" must be cheap enough and "recovery" should be invoked infrequently, in order

Method	**Description**
Residue	Disambiguation based on instruction inspection and residue-based analysis (Section 2.2.3.1 and Section 2.2.3.2, respectively). This is the only method that provides safe disambiguation information.
Region	Corresponds to the application of the speculative region-based alias analysis (Section 4.3.1).
$\text{PG}_{Residue}$	Profile-guided speculative residue-based alias analysis (Section 4.3.2).
PG_{Region}	The profile-guided speculative technique is applied to the also speculative region-based method (Section 4.3.2).
$\text{PG}_{Region+}$	Corresponds to the previous analysis method, but contents of memory cells are assumed to not be used as memory pointers (Section 4.3.1.1).

Table 4.2: Description of may-alias analysis methods for memory disambiguation.

to not incur into unnecessary penalties. Discussion of speculative optimizations as well as check-and-recovery mechanisms are, however, beyond of the scope of this work.

In general, speculative alias analysis is particularly well suited to be used in combination with speculative optimizations based on reordering memory operations [HSS94, GCM+94, MM97, BCC+00, PGM00a]. An example of such optimizations can be seen in Figure 4.8, where different techniques are used for executing load I_2 before the preceding store I_1. By using a new disambiguation state of *"likely independent"* (see Section 4.3.4), our speculative disambiguator not only provides information about which instructions are *likely* to be moved, but also which ones are *not recommended* to be involved in code motion.

4.3.4 Evaluation

In this section, we describe the process we have followed for evaluating the effectiveness of the may-alias analyses proposed in this chapter.

We have implemented the proposed may-alias analysis algorithms, which are presented in Table 4.2, on the same framework we used in Section 4.2.3. As a result, we have obtained a high-quality, low-cost, combined speculative may-alias analysis algorithm for executable code, which uses the following scheme for computing may-alias information:

Phase 1 An interprocedural data-flow analysis computes the *use-def* chains (see Section 2.2.3.1). This phase is the only one required for detecting must-alias information (Section 4.2).

Phase 2 The algorithm performs an interprocedural data-flow analysis computing residue-based information (Section 2.2.3.2).

Phase 3 So far, our algorithm has only computed *safe* alias information, since the analyses we have applied are not speculative at all. Thus, coupled with Phase 2 (i.e., at the

```
Input: Two memory instructions $I_1, I_2$.
Output: An alias relationship {dependent, independent, likely independent, unknown}.
Method:
    if ud-chains($I_1, I_2$,path-insensitive) $\neq$ unknown then
        return ud-chains($I_1, I_2$path-insensitive);
    elsif alias_analysis($I_1, I_2$,safe) $\neq$ unknown then
        return alias_analysis($I_1, I_2$,safe);
    elsif alias_analysis($I_1, I_2$,unsafe) $\neq$ unknown then
        return likely independent;
    else
        return unknown;
    endif
End Method
```

Figure 4.9: Speculative memory disambiguation scheme, for a pair of memory instructions.

same time), we apply the data-flow equations that compute region-based information (Section 4.3.1), which produce preliminary unsafe aliasing data.

Phase 4 Finally, phases 2 and 3 are recomputed (i.e., residue-based and region-based analyses) as speculative profile-guided schemes (Section 4.3.2). Additionally, region-based analysis may now assume that contents of memory cells will not be used as memory pointers (i.e., corresponding descriptors are mapped to $\top$; see Section 4.3.1.1). The resulting data and the one computed in Phase 3 make up the *unsafe* alias information.

We have also used the same execution threshold (i.e., $\phi = 0.6$), and the same *Alto/Inline* binaries as our baseline benchmark set. Memory disambiguation for a particular pair of memory references is then applied incrementally, by looking up the alias information computed for every one of the given phases. Although the above algorithm computes all the necessary alias information, no matter what type of alias data we were interested in, we are only interested in evaluating path-insensitive information. Thus, the corresponding speculative disambiguator is presented in Figure 4.9. Note that a new relationship is used for those pairs of references which are likely to be disjoint. As we advanced in Section 4.3.3, this new status of "likely independent" gives the choice of conscious speculation to any following speculative optimization.

4.3.4.1 Measuring static precision

As we did in Section 4.2.3.1, we start evaluating the effectiveness of each may-alias analysis described in Table 4.2 by comparing their static accuracy in terms of disambiguation queries. In this case, the value returned by the speculative disambiguator presented in Figure 4.9

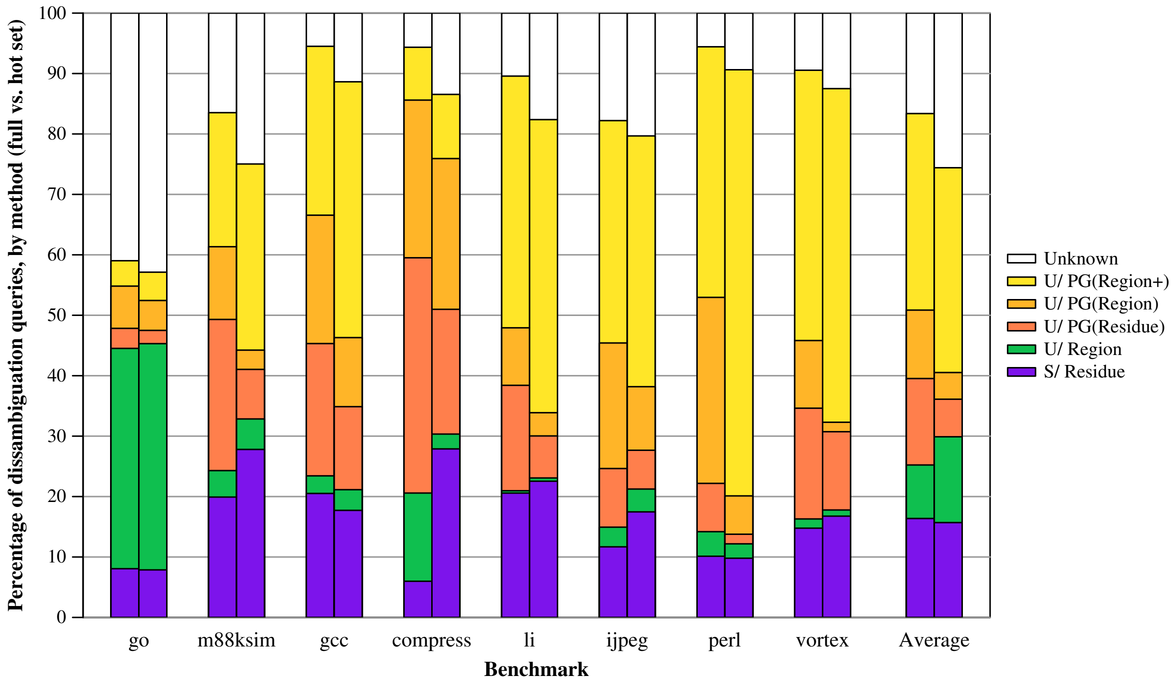


Figure 4.10: Breakdown of disambiguation queries, by path-insensitive alias analysis method. The left bar considers the full set of queries, while the right bar restricts the set to those queries where both components are instructions from *hot* paths. The *S/* and *U/* prefixes denote *safe* and *unsafe* analysis, respectively.

will be *dependent*, *independent*, *likely independent* or *unknown*. We are not really interested in discovering the exact type of relationship (i.e., dependency or independency), but rather in whether the analysis returned an answer different than "unknown". We use the same algorithm presented in Section 4.2.3.1 to generate our set of queries for disambiguation.

For each benchmark, Figure 4.10 presents the percentage of queries successfully resolved by each of the path-insensitive alias analysis presented in Table 4.2, taking into account that the analyses are applied incrementally. For instance, the region-based analysis is not invoked if the residue-based analysis has already successfully resolved the query. To goal is in this case to measure the precision achieved by every one of our speculative may-alias analyses. In the figure, the left bar presents the relative contribution of each analysis to the resolution of our given full set of queries, while the right bar presents the same results restricting the set of queries to those where both components are instructions from *hot* paths. From the results shown in Figure 4.10, four main conclusions can be drawn:

1. Speculative alias analysis is quite beneficial: aliasing precision increases to 83% in average (74% considering only hot path references), from a baseline precision around 16%

corresponding to the non-speculative schemes. Some cases such as `gcc` or `compress` achieve up to 95% of precision. Of course, this spectacular reduction in the number of "unknown" responses will only translate into positive opportunities for optimization if the number of misspeculations (errors made by our analysis) is sufficiently low. The misspeculation rate for each analysis is discussed in the next section.

2. Profiling information proves very useful to increase the accuracy of both the non-speculative residue-based analysis and the speculative region-based analysis. Indeed, the profile-guided analyses almost double the total accuracy achieved by these same methods without using profile information (50% accuracy in front of 26%). An interesting exception is `go`, where the profile-guided schemes do not significantly increase precision. The reason is that, in `go`, almost every execution path is a hot path. Therefore, profile data is not helping to reduce the number of paths to be considered. In general, we note that region-based and profile-guided analysis are orthogonal, and the combination of both schemes achieves more precision than any of them applied separately.

3. The $PG_{Region+}$ (i.e., the heuristic based on the assumption that memory cells will not be used as memory pointers) achieves a high level of precision. This is not very surprising, since our benchmark programs make heavy use of pointers which are naturally stored in memory, while the other alias analyses conservatively assume the worst scenario by setting the descriptor of the corresponding destination registers to $\perp$. For example, program `perl`, which makes heavy use of dynamically linked lists, jumps to a 94% of accuracy from a 53% achieved by the previous analyses. As a result there are many load instructions that indeed read from memory a value that is later used as a pointer, but they are assumed independent by the $PG_{Region+}$ scheme. As we will see in Section 4.3.4.2, although this disambiguation method is extremely aggressive in assuming independence, is also right most of the time.

4. Finally, comparing results for the full set of queries (left bars) versus query results for the hot path only (right bars), it is clear that accuracy for the profile-guided schemes is slightly lower on the hot path. This was expected since profile-guided schemes simply return "likely independent" on those queries where there is an instruction that belongs to a cold path, thus increasing accuracy on the full set of queries.

4.3.4.2 Measuring misspeculation rate

As mentioned in the previous sections, speculating at analysis time will open opportunities for speculative optimizations, which will only be profitable if our guesses are mostly correct. Otherwise, the cost of the particular recovery scheme implemented by the optimization will

Benchmark	**Static misspeculation rate (%)**				
	Region	PG$_{Residue}$	PG$_{Region}$	PG$_{Region+}$	Always
`099.go`	0.00	13.54	13.09	8.97	10.20
`124.m88ksim`	0.00	0.97	1.49	0.80	2.10
`126.gcc`	0.00	1.84	1.96	1.58	2.16
`129.compress`	0.00	1.49	2.07	1.86	2.67
`130.li`	0.00	1.25	1.14	0.65	1.20
`132.ijpeg`	0.00	1.02	1.03	1.36	1.93
`134.perl`	0.00	1.18	2.53	0.91	1.13
`147.vortex`	0.00	0.23	0.64	2.17	4.44
Geometric Mean	0.00	1.37	1.87	1.57	2.48

Table 4.3: Percentage of queries that were misspeculated at least once, relative to the total number of sometime-executed queries.

offset the benefits of our speculative alias analysis. This section presents data on the number of times that each speculative disambiguator produces a wrong answer.

Measuring misspeculation is not an easy task, since misspeculation rate must be measured at run time and depends on the program input data[9]. The process we used is as follows. When running the speculative alias disambiguator in our link-time optimizer, the compiler generates a file including every query whose answer is "likely independent"[10]. Then, we modified the **`sim-profile`** simulator of the SimpleScalar 3.0 toolset [BA97] to read this file at start time and build a hash table with all queries. Every time that a load/store instruction is reached, the hash table is checked to see if the instruction is a component of a query (or queries). If this is the case, we then check if the other member of the query pair has been executed in the past (in dynamic instruction order). If so, we compare their effective addresses, which were stored also in the hash table when each member of the pair was executed. If the effective addresses overlap, we have a misspeculation and we increase the misspeculation counter for that particular query. In any case, the total execution counter for the query is also incremented. Care must be taken to ensure that as we revisit a given instruction, we do not increment the counters if the other member of the query pair has not been previously executed also. At the end of the simulation run, we have for each query the number of times it was dynamically executed and the number of times it was misspeculated.

From this procedure we obtain two sets of results, presented in Table 4.3 and Table 4.4. First, Table 4.3 shows the number of queries that were misspeculated *at least once*, presented as a percentage of the total number of queries that were sometime executed. However, to get a complete picture of the cost of misspeculation, we need to know *how many times* a

[9]For our experiments, we used our benchmark suite with the simulation inputs presented in Table 3.2.

[10]We chose only queries where both components are *hot* instructions, since the rest of possible queries will be rarely executed at run time.

Benchmark	**Dynamic misspeculation rate (%)**				
	Region	$\mathrm{PG}_{Residue}$	PG_{Region}	$\mathrm{PG}_{Region+}$	Always
`099.go`	0.00	1.60	0.98	0.70	1.14
`124.m88ksim`	0.00	0.02	0.92	0.32	8.42
`126.gcc`	0.00	2.32	1.80	1.02	1.93
`129.compress`	0.00	0.20	3.05	2.48	2.73
`130.li`	0.00	0.46	0.82	0.75	1.01
`132.ijpeg`	0.00	0.73	0.96	2.36	3.21
`134.perl`	0.00	0.81	1.66	0.70	1.49
`147.vortex`	0.00	0.04	0.15	1.99	2.21
Geometric Mean	0.00	0.34	1.00	1.04	2.20

Table 4.4: Percentage of dynamic queries misspeculated, relative to the total number of dynamic queries.

misspeculated pair of instructions was executed to know the number of times that the associated recovery scheme would have been invoked. Table 4.4 presents this second set of data, showing the *total* number of misspeculations, presented as a percentage over the total number of dynamically executed queries. The last column in both tables corresponds to an "aggressive disambiguation" approach assuming that "unknown" query responses will be always handled as if they were "likely independent". This will give a measure of whether the proposed speculative alias analysis methods are useful for speculative optimizations.

From the results presented in the above tables, we can highlight several interesting points:

- Region-based analysis, despite being unsafe, is extremely accurate, to the point that we did not find a single misspeculation across all benchmarks. This result encourage the claim we made in Section 4.3.1.2 about using this analysis as a safe analysis on user-demand, like actual production compilers do on several unsafe optimizations.

- The other speculative analysis are fairly accurate too, with a static misspeculation rate typically below 2%. Measured in terms of dynamic misspeculation rates, as Table 4.4 shows, the situation is even more favorable. For example, for program `go`, even though more than 8% of the static queries were misspeculated, their weight over the total number of executed queries is much lower, typically below 2%.

- We mentioned in Section 4.3.4.1 that $PG_{Region+}$ is based in a risky assumption about memory cells not being used as memory pointers. However, the results show that, although being very aggressive in assuming independence, this scheme is right most of the time. How can be $PG_{Region+}$ so accurate? The key insight is that pointers are hardly ever aliased to other pointers in a program. Again, a good example is program `perl`, which makes heavy use of dynamically linked lists whose elements are hardly ever aliased to each other.

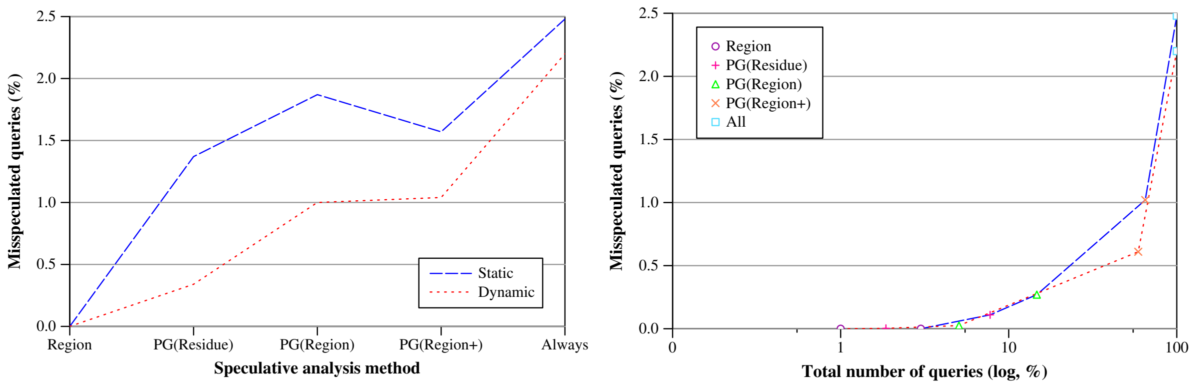


Figure 4.11: Percentage of misspeculated queries for every alias analysis methods, with respect to (a) number of speculated queries for every method, and (b) total number of possible speculated queries (this one also shows the percentage of queries covered for every method).

- Finally, when looking at the "always speculative" approach, the data shows that misspeculation rate doubles the rate obtained using our most aggressive speculative method (i.e., $PG_{Region+}$; a geometric mean rate of 2.2% in front of 1.04%). This fact is encouraging, hinting that even for those optimizations with high-cost recovery techniques, our speculative alias analysis might prove very useful given its high accuracy.

For the percentages presented in the above tables, it's important to note that the total number of static/dynamic queries for every analysis method (i.e., columns) is different in every case, and depends on the number of queries for what such particular disambiguation method returned the known "likely independent" answer. As a result, we are comparing relative percentages which are obtained considering different "base" numbers. The reason is that we are interesting in measuring the relative misspeculation of every disambiguation method due to only those answers given by such method in any case. Figure 4.11a shows in a graph the geometric means of these percentages, which were already presented in Table 4.3 and Table 4.4.

A different way of showing the misspeculation rate of our proposed analyses is shown in Figure 4.11b. In this case, we present in the x-axis the percentage of the total possible number of speculated queries (i.e., obtained from the *Always* scheme) covered for every alias analysis method, while the y-axis shows the misspeculation rate relative to such total number of queries. We can see that coverage of the earlier disambiguation methods is quite low, which turns out that speculation in these cases is somehow conservative. However, we can see for example that dynamic misspeculation rate of the $PG_{Region+}$ is only about 0.6%, with a coverage of 60% of queries. This means that the remaining 40% of dynamic queries is responsible of the remaining 1.5% misspeculation rate, which give us a definitive measure of confidence about the accuracy of our disambiguation schemes.

4.4 Related work

While there is an extensive body of work on pointer alias analysis of various kinds [WL95, SH97, DWM98, CH00, GLS01], these are mostly high-level analyses carried out in terms of source language constructs that turn out to be of limited utility at the machine code level. In fact, to the best of our knowledge, alias analysis carried out by existing object code modification systems is often limited to fairly simple local analysis.

Amme *et al.* [ABZT98] present a general method to detect data dependencies in assembly code by using symbolic value propagation. They are able to provide both must- and may-alias data, which allows deriving memory dependencies, but this information is only provided in a path-insensitive fashion. Besides, the algorithm does not work beyond procedure boundaries, and symbolic values are not propagated through memory when registers are saved/restored. Although it has been applied to assembly code, it is not obvious that using the algorithm for interprocedural whole-program analysis would scale up to problems of this size.

Our work is based on the work of Debray *et al.* [DMW98], that has been described in Section 2.2.3.2. They propose a flow-sensitive, context-insensitive interprocedural *may*-alias analysis algorithm designed in the context of a link-time optimizer, that attempts to deal with features of real executable programs. However, the algorithm fails in several situations, which turns out to lead to an undesirable loss in precision. First, it is clearly oriented to "fine grain" disambiguation, but it is unable to effectively catch "coarse grain" alias relationships as we do in Section 4.3.1. Second, for keeping the algorithm space and time feasible, they apply a conservative widening operation that maps to $\perp$ definitions of the same register from different control-flow paths. Finally, they do not keep track of the contents of memory when registers are saved/restored. Every one of these drawbacks have been somehow addressed in this work, although we have use a combination of speculative unsafe schemes to succeed.

There is a considerably body of work on interprocedural data-flow analyses design to analyze only part, but not all, of a program (see, for example, [AL98, BA98, DGS97, RRL99]), although only some of them use profile information to guide their decisions. This profile information is, however, widely used when performing optimizations [PH90, CMCH92, CL96, GBF98]. On the other hand, while speculation has been commonly used in the compiler world for optimizing programs [HSS94, GCM$^+$94, MM97, GBF98, PGM00a], as far as we know, this is the first attempt to introduce unsafe speculations into a data-flow analysis algorithm.

Finally, different alternatives for collecting profile information in order to detect memory independences have been proposed [Con97, CFE97, RCT$^+$98], but the cost of the instrumentation process is usually high, and not general enough. By contrast, our approach allows to discover "important" memory independencies just using simple basic block counts easy to collect by simple basic block instrumentation.

4.5 Conclusions

Code transformations on executable code can benefit greatly from pointer alias information. However, the problem of memory disambiguation is one of the weak points of object code modification, because important high-level information typically available in a traditional compiler is lost at executable code level. Besides, existing alias analyses turn out to be of limited utility at the machine code level, because either they do not consider typical issues of binary code, or their application is too expensive to be practical for analyzing large binaries.

This chapter has presented several approaches for high-quality alias analysis to be applied in the context of link-time or executable code optimizers. First, we propose a new high-accurate *must*-alias analysis to recognize memory dependencies in a *path-sensitive* fashion. The analysis is based on the idea of establishing alias relationships for only a subset of all the possible paths between every pair of references to disambiguate. Our results have shown that by using this technique we are able to increase alias precision up to 150% over a path-insensitive baseline scheme. Thus, this alias information is particularly well suited to be used for eliminating path-sensitive redundant memory operations, as we will see in Chapter 5.

We also have presented two approaches for high-quality, low-cost, speculative *may*-alias analysis to recognize memory independencies. The key idea behind our two proposals is to trade off analysis complexity against *safeness*. Thus, we presented a region-based alias analysis that disambiguates memory references by classifying them into separate memory regions (e.g., heap, stack and global) and assumes that whenever an arithmetic operation is performed on a pointer, the pointer will not change its pointed-to memory region. On the other hand, a second speculative proposal uses profile information and assumes that instructions on hot paths are not aliased to memory references in cold paths. These assumptions, although unsafe, yield significant increases in disambiguation accuracy to over 80%, while the run-time misspeculation rate is below 2%. Although it has been commonly used for optimizing programs, speculation has been only introduced so far in the compiler world at analysis time by means of a set of rules and heuristics. As far as we know, the work presented in this chapter is the first attempt to systematically introduce unsafe speculations into data-flow analysis algorithms.

The alias algorithms we have presented are targeted to provide both *must*- and *may*-alias information. Our results show that these algorithms prove to be very useful for increasing memory disambiguation accuracy of binary code, which turns out into opportunities for applying optimizations such as eliminating binary redundancy.

Chapter 5

Memory redundancy elimination

In this chapter we reason about the first source of binary redundancy targeted in this thesis: we discuss the discovery and elimination of redundant memory operations in the context of a link-time optimizer, an optimization that we call Memory Redundancy Elimination (MRE)*. We first motivate this work by measuring a potential upper bound on how much memory redundancy is present in executable programs. Then, we present a set of sophisticated profile-based MRE algorithms targeted at optimizing away these redundancies. Finally, we provide some experimental results, including accurate measurements at the processor-core level, which demonstrate that MRE yields significant performance improvements when applied at executable code level.*

5.1 Introduction

Without comparison, caches are the best hardware approach to address the memory bottleneck [Smi82]. As we pointed out in Section 1.1.1, one of the reasons why caches are effective is because they *reuse* recent memory accesses. That is, they exploit the *dynamic memory redundancy* existing in programs. In the ideal case, the compiler can also benefit from these reuse opportunities by promoting repeatedly accessed memory locations to registers (see Section 2.3.3.2). There are several reasons that explain why this is an important optimization:

- Memory operations are expensive to execute in parallel, because they require multiple ports to the hardware cache [Smi82]. In comparison, an access to the register file is a much cheaper operation.
- Most compiler transformations break up their optimization scope when detecting memory accesses. Meanwhile, eliminating memory instructions enables other optimizations, thus decreasing the dynamic operation count and instruction schedules.
- There are many redundant memory operations at run time, as our experiments will show, and many of them can be removed.

Memory redundancy comes partly from the way that programmers write source code. However, as we advanced in Section 1.1.1, a significant number of redundant memory references appear in the final executable file due to limitations in the compilation model of traditional compilers. Thus, for example, a variable may not have been kept in a register by the compiler because it was a global, or maybe the compiler was unable to resolve aliasing adequately, or because there were not enough free registers available.

To address the above issues, we propose in this chapter an optimization to be applied in the context of binary or link-time optimizers. We discuss the discovery and elimination of memory operations that are redundant and can be safely removed in order to speed up a program, an optimization that we call *Memory Redundancy Elimination (MRE)*. First, we quantify how much memory redundancy is present in executable programs, and show that a high percentage of memory references can be considered redundant because they are accessing memory locations that have been referenced in the near past. Then, we present several profile-based MRE algorithms targeted at optimizing away these redundancies:

- A basic MRE algorithm for extended basic blocks.
- A general MRE algorithm that works over regions of arbitrary control-flow complexity, for removing either fully or partially redundant loads and stores.

- A new technique for eliminating redundant loads in a path-sensitive fashion, since the above optimization algorithms are mainly based on path-insensitive PRE techniques that causes many MRE opportunities to be lost.

- A simple set of heuristics for removing dead stores.

We also provide some experimental results in order to measure the effectiveness of the MRE algorithms under study. Our results show that a significant amount of memory redundancy can indeed be eliminated, which translates into important reductions in execution time.

5.2 Dynamic memory redundancy

Before presenting our algorithms for removing memory redundancy, we motivate our work by measuring a potential upper bound on how many memory instructions could be removed from a program. To achieve this goal, we run on top of the SimpleScalar `sim-profile` simulator our baseline benchmark set with the corresponding simulation inputs (see Section 3.2.2 and Section 3.3, respectively) to capture every dynamic memory reference. Dynamic memory redundancy is then measured by recording the most recent n memory references into a *redundancy window*. This window is a simple FIFO queue, where new references coming into it displace the oldest memory reference stored in the window. We next present the observed results for dynamic load and store redundancy.

5.2.1 Dynamic load redundancy

As far as load redundancy is concerned, our goal is to measure how often a load is re-loading data that has already been either loaded or stored in the near past, and also to quantify the typical distance (in memory instructions) between re-loads of the same data item. A dynamic instance of a load is then redundant if its effective address matches the address of any prior load or store that still remains in the redundancy window.

The results of our experiments are shown in Figure 5.1a, where we present data for our original benchmark set and for various redundancy window sizes. Clearly, a lot of redundancy exists even in these highly optimized binaries[1]. As an example, the graph shows that, for program `vortex`, almost 75% of all load references were to memory locations that had been referenced by at least one of the most recent 256 memory instructions. That is, almost 75% of all load references were to memory locations that had been either loaded or stored recently. In general, almost 50% of all loads are re-loading a data item that was read/written less than 100 memory instructions ago. Considering that in these streams around 1/3 of instructions

[1]As we mentioned in Section 3.3, programs were compiled with full optimizations using the Compaq/Alpha C compiler. Similar levels of redundancy have been observed using other compilers, like GNU `gcc`.

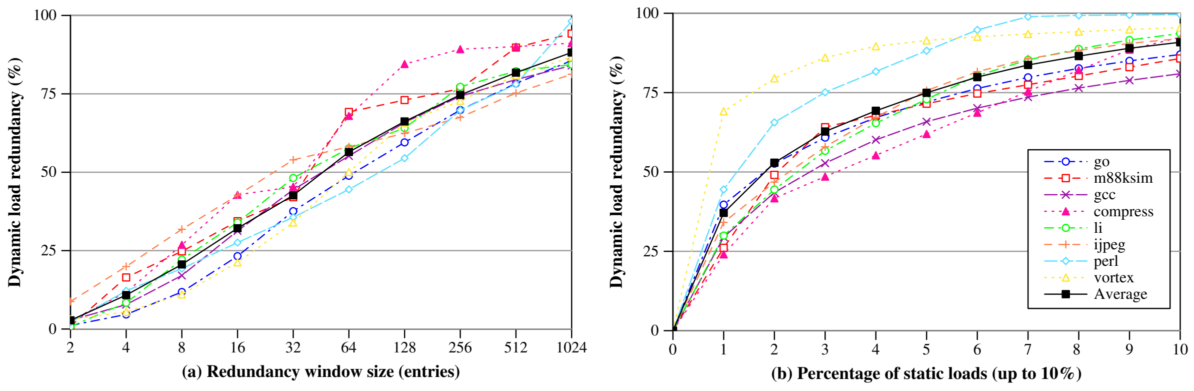


Figure 5.1: Dynamic amount of load redundancy vs.: (a) redundancy window size (X-axis is logarithmic), and (b) percentage of static loads for a 1024-entry redundancy window (X-axis is only drawn up to 10%).

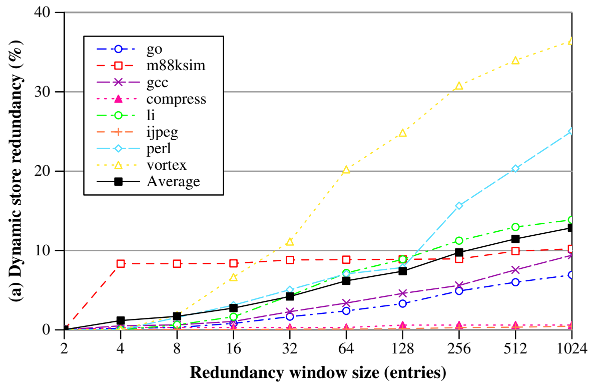
are memory references, it means that 50% of all loads are re-loading data that was already accessed or stored less than 300 instructions ago. Today's optimizing compilers are clearly able to deal with regions larger than these sizes and, thus, should be expected to optimize all this redundancy away.

Figure 5.1b shows the redundancy observed for each program by accumulating the contribution of every static load (sorted by decreasing redundancy), under a redundancy window size of 1024 entries. The results show that more than 75% of the dynamic load redundancy observed is caused by less than 8% of the load references. This fact is encouraging, hinting that MRE techniques might have a great impact if they were able to remove some of the most redundant load references.

5.2.2 Dynamic store redundancy

A dynamic instance of a store is considered redundant only if (a) its effective address matches the address of a new store while it still remains in the redundancy window, and (b) there is no load instruction between them accessing the same location. Figure 5.2a shows the dynamic store redundancy exposed by every program in our benchmark suite, as was done in Figure 5.1a when measuring dynamic load redundancy.

The first thing to observe is that, unlike load references, most store instructions are not redundant at all, which is consistent to observations made by Zhang [Zha02]. Even considering a big redundancy window of 1024 entries, the experiment shows that, except for programs `perl` and `vortex`, only around 12% of store redundancy is observed. This can be explained looking at the results presented in Figure 5.2b, which shows the dynamic *write-after-read* (WAR) memory dependency rate (that is, when a store matches the address of a previous load in the redundancy window with no other matching store between them). As we can



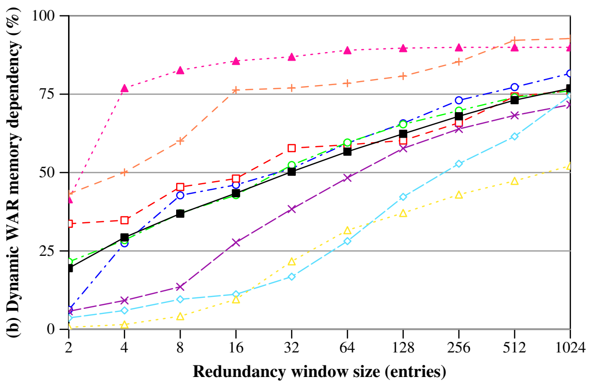


Figure 5.2: Dynamic amount of (a) store redundancy, and (b) write-after-read memory dependencies, vs. redundancy window size (X-axis is logarithmic).

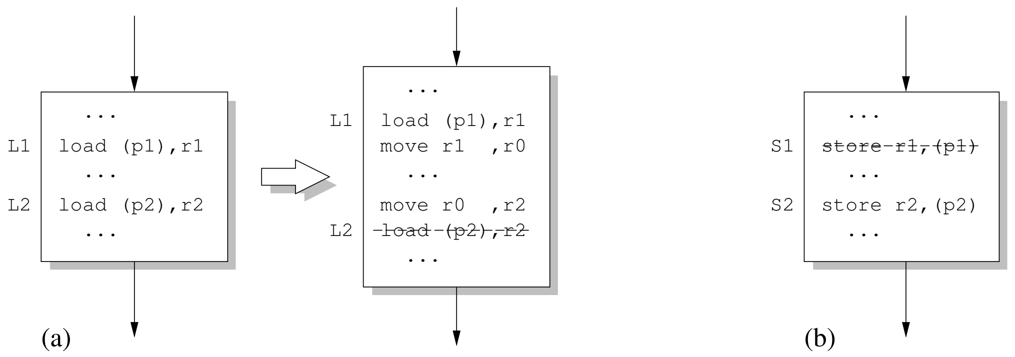
observe, around 75% of the store references are writing to a memory location previously read by a load without any intervening store instruction involved. Consequently, even considering such a big window, there are fewer opportunities for removing redundant stores.

5.3 MRE on executable code

The simplest examples of *Memory Redundancy Elimination (MRE)* are shown in Figure 5.3. First, let's consider the case of eliminating a redundant load in Figure 5.3a, which we call *Load Redundancy Elimination (LRE)*. Suppose that an instruction L_1 loads a value into register r_1 from memory location pointed by register p_1. Furthermore, this load is followed after some instructions by another instruction L_2 within the same basic block, which loads a value into register r_2 from location pointed by register p_2. If it can be proven that p_1 and p_2 point to the same location, and that this location is not modified between these two instructions, then L_2 is *redundant* with respect to L_1. Note that redundancy would be also present if instruction L_1 was a store operation.

Once a redundant load has been identified, we may try to eliminate it by *bypassing* the value from the first load to the redundant one. Bypassing the value is accomplished by inserting a couple of move operations that use a new *available* register r_0, which may or may not be the same as r_1 or r_2 depending on register lifetimes. The expectation is that, after running the LRE optimization, a copy propagator is also run to eliminate as many register moves introduced by LRE as possible.

The case of eliminating a redundant store, also called *Store Redundancy Elimination (SRE)*, is shown if Figure 5.3b. In this case, an instruction S_1 that stores a value into location pointed by register p_1 is followed by another instruction S_2 within the same basic block that stores another value into location pointed by register p_2. If, as before, it can be



```
   ...
L1 load (p1),r1
   ...
L2 load (p2),r2
   ...
```

```
   ...
L1 load (p1),r1
   move r1  ,r0
   ...
   move r0  ,r2
L2 ~~load (p2),r2~~
   ...
```

```
   ...
S1 ~~store r1,(p1)~~
   ...
S2 store r2,(p2)
   ...
```

Figure 5.3: Elimination of a redundant memory reference within a machine code basic block: (a) Load Redundancy Elimination (LRE), and (b) Store Redundancy Elimination (SRE).

proved that both locations pointed by p_1 and p_2 are the same, and this location is not loaded between these two references, then S_1 can be safely removed because it is *redundant* with respect to S_2. Unlike LRE, there is no need for bypassing values, since eliminating S_1 does not alter the program semantics.

Although these are the most simple cases of MRE, they already introduce the three fundamental problems that this optimization has to deal with:

Alias analysis The first problem is to decide if any load/load or store/load (store/store) pair is really accessing the same memory location or not, and also to prove that there is no other store (load) between them that *may* be in conflict with the memory location accessed by the redundant load (store). In our example in Figure 5.3, this amounts to proving that registers p_1 and p_2 do indeed point to the same memory location. Although there is an extensive work on pointer alias analysis (see Section 2.2.3), these are mostly high-level analyses carried out in terms of source language constructs that turn out to be of limited utility at the machine code level.

Register liveness analysis When applying LRE, the second problem is to find an available register to bypass the value from the redundancy source to the redundant instruction. This is not an easy task, due to the limited number of machine registers and also due to the constraints imposed by the calling convention. Register liveness analysis (see Section 2.2.1) is a technique that computes which registers are alive at every point in the code. On executable code, control-flow reconstruction is key to improve the accuracy of the liveness analyzer. Otherwise, the analysis becomes too conservative to be useful.

Cost-benefit analysis Finally, also for LRE, the example at Figure 5.3a shows that eliminating the load does not come without a cost. In fact, we have inserted two "move" instructions in the optimized code in the hope that (a) they can be removed by a copy propagator and (b) even if they are not, their cost will be lower than that of the original redundant load. Of course, the cost can be reduced if we can use register r_1 as the bypassing register. This, however, will require that r_1 is not overwritten between instructions L_1 and L_2. In any case, LRE on executable code requires of a careful cost-benefit analysis, as Section 5.4 will discuss. If the cost-benefit analysis is too optimistic, performance degradation may result.

Alias and register liveness analysis are well-known data-flow problems already described in the literature [ASU86a, Muc97f]. From now on, we assume that both of them have been computed before applying MRE optimizations. The more accurate these analyses are, the more opportunities appear for MRE. A significant number of opportunities may be lost if the alias analyzer is not able to decide whether two references are in conflict. Also, discovered LRE opportunities are lost if the register liveness analyzer is not able to find an available register to effectively bypass the redundant value.

5.3.1 MRE on intermediate vs. executable code

It is interesting to point out the differences between performing MRE on intermediate code, as done by most compilers, and on executable code. Although the process is similar in both cases, some issues must be managed in a different way.

First, most compilers will perform LRE by taking a new "pseudo-virtual" register from the infinite virtual register pool to bypass the value between the redundant pair. Interestingly, it may happen that at a later stage, when the register allocator runs, the compiler re-inserts the redundancy due to lack of machine registers. Performing LRE on executable code does not suffer from this problem, since estimating the costs and benefits of inserting and removing instructions is rather more accurate than when working on an intermediate representation. However, our proposed optimization must deal with the limitations of a small register file.

Working on executable code also has the added difficulty that, in general, no information on the original program variables is available. As a result, not only alias information becomes imprecise, but also signal handling and volatile variables may pose correctness problems when applying MRE. A common solution is to disable this or other link-time optimizations via compiler switches, when optimizing programs that contain such constructs. However, a preferable solution for users would be that compilers insert information about volatiles into the executable program, so that a link-time optimizer could use this information to only avoid applying MRE on such variables.

Finally, as we pointed out in the previous section, LRE on executable code requires of a careful cost-benefit analysis, since we assume that the cost of inserting move instructions will be lower than that of the original redundant load. If this cost-benefit analysis is too optimistic, performance degradation may result.

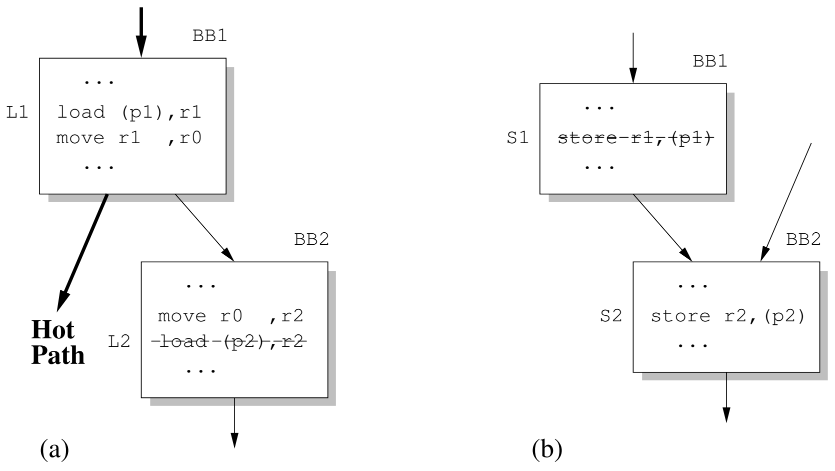


Figure 5.4: Elimination of redundant memory references within extended basic blocks applying (a) LRE, and (b) SRE. LRE should be applied coupled to a cost-benefit analysis.

5.4 Profile-guided MRE

Information about the program execution behavior can be very useful in optimizing programs (see Section 2.3.2). The key idea is to be aware of *profile information* to guide MRE. Profile information consists of a frequency for each basic block and a probability for each branch in the program. We next outline the algorithms used and present their associate cost-benefit equations, which use the basic block frequency information gathered in a profile run to choose the candidates for removal.

5.4.1 Eliminating close redundancy

The results presented in Section 5.2 show that between 20% and 40% of the load redundancy (up to 8% of the store redundancy) detected can be captured using a redundancy window of just 16 entries. This indicates that the first source of redundancy that we should target our optimization at is located within small groups of basic blocks.

We have already seen the easiest forms of MRE in the example given in Section 5.3, where we look for redundancy within a basic block. A natural extension of this scheme is to perform MRE on an *Extended Basic Block (EBB)*, as they were defined in Section 2.1. In particular, LRE will be performed on EBBs, while reverse EBBs will be used for applying SRE. The implementation is in this case straight-forward by performing a linear search to the EBB root, as shown in Figure 5.4.

When applying LRE in Figure 5.4a for every load in the EBB, we search bottom-up for other loads or stores that may be a source of redundancy. If we can prove that registers p_1 and p_2 point to the same memory location and that no intervening store has modified said location[2], then it is safe to remove L_2 and bypass the value from r_1 to r_2. Similarly, for applying SRE in Figure 5.4b, a top-down search through the reverse EBB is performed for every store, looking for other stores writing the same location with no intervening loads between them. If this is the case, the considered store S_1 can be eliminated safely.

As already discussed in Section 5.3, introducing a move instruction on LRE increases the cost of executing basic block BB1. What if, as in Figure 5.4a, the hot path does not flow through BB2? In this case, a move instruction has been inserted in the common path, although the bypassed value will be most often discarded. There is no benefit in applying LRE to L_2 in this scenario and we might risk lowering performance. The lesson to learn is that it is not always beneficial to remove a redundant load, and it is necessary to apply LRE carefully. We need to compute as precisely as possible the benefit (B) and cost (C) of applying the optimization for each particular load. The equations we use are as follows:

$$\begin{aligned} B &= lat_{load} \times BB_2^{freq} \\ C &= lat_{move} \times (BB_1^{freq} + BB_2^{freq}) \\ LRE &\Leftrightarrow C < B \end{aligned} \tag{5.1}$$

The use of a *less-than* comparison as opposed to a *less-or-equal* comparison in the trade-off between cost and benefit is deliberate, since it avoids unnecessary code motion. As it can be seen, both benefit B and cost C computations consider the latency of the involved instructions weighted by their execution frequencies. Note that C is pessimistic, as it includes both move instructions, even though they might be later removed by the copy propagation phase. Our current implementation checks first whether either the source of redundancy register or the final destination register (i.e., r_1 and r_2 in our example, respectively) can be chosen to bypass the redundant value, avoiding some of the move insertions and keeping the cost more realistic.

5.4.2 Eliminating distant redundancy

The MRE approaches described in the previous section were targeted at exploiting close redundancy. However, looking back to Figure 5.1a and Figure 5.2a, there is still a lot of redundancy that can be caught if we could explore larger distances between instructions. Of course, in order to catch this redundancy, we need to apply MRE to regions of code that expand beyond an EBB and which, therefore, may contain complicated control-flow structures.

The major difference with the previous section is that when working on a candidate reference (load or store) to be removed, we need to examine *all* the possible control-flow paths that may reach the candidate in order to decide whether it is truly redundant or not. Besides,

[2]If an intervening store can be proved to write to the same location, then it becomes itself the source of redundancy and the algorithm works the same way. The problem is when an intervening store has an unknown address. In such case, bypassing is not safe and redundancy elimination can not proceed.

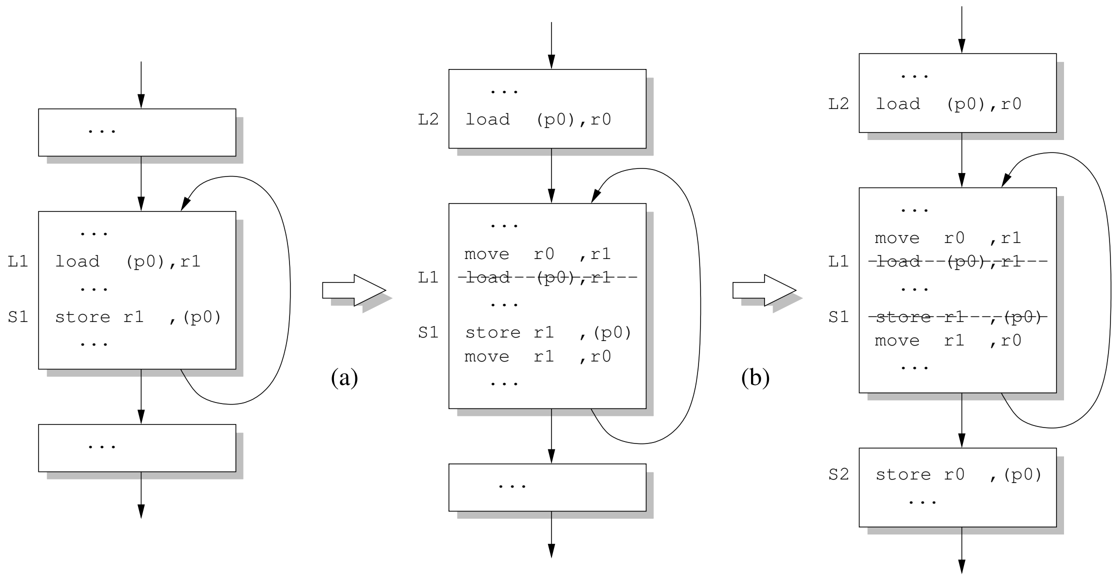


Figure 5.5: Elimination of (a) partially redundant load, and (b) partially redundant store. Removing the redundant references requires inserting instances on less-frequent paths, in order to make the candidates fully redundant.

this redundancy will come usually from severals points, or redundancy sources. Two different situations may arise:

Full redundancy The candidate is redundant with respect to all the control-flow paths that reach it. As always, all intervening stores for LRE and loads for SRE must have known addresses that do not alias with the candidate. We call *full-MRE* the optimization targeted to remove this type of redundancies. Removal in this case is a safe transformation, because there is always at least a static source of redundancy for every candidate.

Partial redundancy The candidate is redundant on *some* paths, but not all, that reach it. Actually, a high percentage of dynamic redundancy comes from candidates that are redundant only on some control-flow paths. For example, in Figure 5.5 instruction L_1 is redundant on the loop-back edge with the store S_1, but it is not on the entry point of the loop (suppose that neither register p_0 nor location pointed by p_0 are changing in the loop). This usually happens on loop invariant variables, but similar situations arise frequently even without considering loops [PGM00b]. We call these candidates *partially redundant references*, and they must be removed by inserting a copy of the instructions on the control-flow paths where they are not available, thus making them fully redundant. In the example, this means that copies of the load and the store must be inserted in both loop pre-header and loop post-tail, respectively. We call *partial-MRE* the optimization targeted to remove partially redundant references.

> Partial-MRE involves insertion of new instructions. As these insertions are usually done beyond EBB boundaries, the inserted instructions become *speculative*. In general, it is safe to perform speculation for instructions that cannot cause exceptions, but this is not the case for speculative memory operations. When speculating references, the optimizer must be careful not to introduce side-effects into a program that did not exhibit them before. However, as our approach is based on standard PRE methods, correctness is warranted since new insertions are always *anticipable* at the given insertion points[3].

Partial-MRE *conceptually subsumes* the behavior of full-MRE, since the only difference relies on whether insertion of new memory references is allowed or not. Thus, both situations can be addressed by using the same algorithm, simply discarding partial redundancies in the case were we are only interested in removing fully redundant references. Note, however, that as both optimizations are constrained by limited resources (e.g., register availability, cost-benefit analysis, etc.), the number of "strict" full redundancies removed may be different in every case.

5.5 Partial MRE

We next present in some depth the details and issues to be addressed when applying both partial-LRE and partial-SRE optimizations, which have been implemented into two separate algorithms. Furthermore, as these optimizations are expensive in terms of compilation time, they should be applied carefully: either (a) no more than once for every candidate, or (b) several times for the most important candidates (i.e., those candidates located in the most frequently executed paths of the program).

5.5.1 Partial LRE

For the implementation of LRE targeted at distant redundancies we have followed the approach described by Horspool and Ho [HH97b]. They proposed a general profile driven PRE algorithm based upon edge profiles. The main idea is to insert copies on less frequently executed paths in favor of more frequently executed paths, as was shown in Figure 5.5. We have adapted their algorithm to only consider redundant load operations.

Once a candidate load has been discovered by the algorithm, we apply our cost-benefit analysis (i.e., Equations 5.1) described in Section 5.4.1. However, we have to extend the cost equation C to account for the insertion of (a) every move instruction on each of the redundancy paths, in order to bypass the source value(s), and (b) the new load operations

[3]Insertion of an expression e at program point p is allowed only if all control-flow paths emanating from p evaluate e before any operands of e is redefined. The expression e is said to be *anticipable* at point p [HH97b].

that make the candidate load become fully redundant, as shown below:

$$\begin{aligned} C_{bypass} &= lat_{move} \times \left(BB^{freq}_{red} + \sum_{i=1}^{n} BB^{freq}_{src_i} \right) \\ C_{insert} &= lat_{load} \times \sum_{i=1}^{m} EDG^{freq}_{i} \\ C &= C_{bypass} + C_{insert} \end{aligned} \tag{5.2}$$

where n is the number of partial redundancies and m the number of load insertions needed. Note that there is no need to consider move operations at new insertion points, since the loads inserted will set the appropriate register. Again, this cost is a pessimistic upper bound, since an appropriate choice of bypass registers may avoid some of the move instructions. As already discussed in the previous section, the load will maintain its “candidate” status only if the benefits of removing it out-weights the computed cost.

When the removal of a candidate load has been considered beneficial, our algorithm starts looking for the best available register to bypass the source value(s). First of all, we start checking whether the destination register can be used to bypass the value from *all source paths* and *all insertion points*. If the destination register can not be used, the basic blocks that are the source of the redundancy are sorted according to their execution frequency. Now, we start with the most executed basic block and check whether we can use the source redundancy register to bypass the value on *all the other* paths and *all insertion points*. We iterate for every source basic block until such register is found. If after this process no register is found, then we simply look for any *free* register that might be used on all paths simultaneously. Note that this would match the pessimistic cost analysis outlined above. If still no register is found, then the redundant load can not be removed.

As far as the insertion points are concerned, the algorithm needs to be able to “materialize” pointers corresponding to the location redundantly accessed. This means that we need to extend the above algorithm for checking register availability not only for bypassing redundant values, but for computing target addresses as well. To this end, our algorithm follows a simple heuristic that allows pointer materialization by adding an offset from a base register, which must be alive at some point in the program for every considered insertion point. In general, materializing pointers is easy for global and stack references, but some opportunities for load removal may be lost when the algorithm fails to materialize a pointer.

5.5.2 Partial SRE

Our algorithm for elimination of redundant stores is also based in Horspool and Ho’s approach. However, a new formulation of this method is needed for dealing with stores[4]. While partial-

[4]Actually, our definition of *redundant store* is known in the literature as *dead store*, which means that we need to develop a *partial dead store elimination* algorithm. We use a different name in this case to distinguish between this type of redundancy and the one we will introduce in Section 5.6.2.

LRE requires forward availability and insertion of loads as late as possible, a partial-SRE algorithm will require backward anticipability and insertion of stores as early as possible. The resulting analysis is next fully described in Section 5.5.3.

Unlike the local approach introduced in Section 5.4.1 for removing stores, the insertion of new store operations will require to bypass, from the redundancy point, the value to be written in memory. To this end, we used a similar algorithm as the one described for the partial-LRE case. The same approach is also used for materializing effective addresses of the new inserted store operations.

5.5.3 A cost-benefit formulation for partial SRE

In this section we describe the formulation for *partial store redundancy elimination* that we advanced in Section 5.5.2, which is based on the Horspool and Ho's approach for general partial redundancy elimination [HH97b]. We assume that complete information regarding the execution frequency of every edge in the flow graph is available. The goal is to minimize the number of times a store instruction is executed (which is the cost measure) given this profile information. The analysis has three phases:

1. Determining the lowest cost of making a store fully redundant at various program points throughout the flow graph, and recording how to achieve such lowest cost.

2. Checking each basic block which contains an instance of the given store instruction, and determining the net benefit achieved if that instruction were to be removed. This phase provides as a result: (a) a set of basic blocks containing store operations that should be deleted, and (b) a set of edges where new instances of the store should be inserted.

3. Finally, checking for register availability in order to compute the final cost of removing the store instruction, which will be done only if still yields some benefit.

In order to simplify these analyses, we consider occurrences of a store s to a particular memory location in a flow graph.

5.5.3.1 Cost analysis

In the fist phase we want to determine (a) the lowest cost (measured as the number of dynamic evaluations) of making s fully redundant at a program point p, and (b) how that lowest cost can be achieved. Making s fully redundant may require performing insertions of s, either at point p itself or in the paths that follow from p[5]. This set of insertions is then known as the *cost set*. The initial conditions on which our analysis is based are as follows:

[5]We will consider insertions of s along flow graph edges only, because this is more general than allowing insertions only in basic blocks. Later, a simple optimization will merge the resulting fake basic blocks.

- Let $TRANSP_i$ be *true* **iff** basic block i is *transparent* to store s (i.e., basic block i does not contain any potential read to location pointed by s).
- Let $ANTIC_i$ be *true* **iff** store s is *locally anticipable* on entry to basic block i (i.e., basic block i contains an instance of store s that is not previously read).
- Let $AVAIL_i$ be *true* **iff** store s is *locally available* on exit from basic block i (i.e., basic block i contains an instance of store s that is not subsequently read).
- Let $cost : \mathbb{S} \mapsto \mathbb{N}$ be a function that maps a cost set to its numeric cost, measured as the number of evaluations of the store s that must be added to the flow graph. Let also $FREQ_{ij}$ be the execution frequency of edge (i, j). Then function $cost$ is defined as:

$$cost(S) = \begin{cases} \infty & \text{if } S = \perp \\ \sum_{(i,j)\in S} FREQ_{ij} & \text{otherwise} \end{cases} \tag{5.3}$$

  Note that a special cost set denoted as $\perp$ is required, which is an upper bound on all cost estimates. It represents a complete lack of information about a solution.

We wish to determine the following cost sets associated with basic blocks in the flow graph:

- Let Cin_i be the cost set on entry to basic block i. A solution $cost(Cin_i)$ for this set can be interpreted as the minimum cost of making a store instruction s fully anticipable on entry to block i.
- Let $Cout_i$ be the cost set on exit from basic block i. The meaning of this set is analogous to the Cin_i set.

These sets are then related by using a conventional system of data-flow equations. The equation for Cin_i is as follows:

$$Cin_i = \begin{cases} Cout_i & \text{if } TRANSP_i \bullet \overline{ANTIC_i} \\ \emptyset & \text{if } ANTIC_i \\ \perp & \text{otherwise} \end{cases} \tag{5.4}$$

The first case of the equation says that if basic block i is transparent to store s and does not contain a locally anticipable instance of s, then we can make s anticipable on entry to block i by making it anticipable on all outgoing edges from i. However, if block i contains an instance of s that is anticipable on its entry, then Cin_i will be an empty set, which explains the second case. This is because no insertions of s are needed to make s fully anticipable at this program point. Finally, the third case means that no insertions anywhere could make s anticipable at this point.

The equation needed for $Cout_i$ is the following:

$$Cout_i = \begin{cases} \bot & \text{if } i \text{ is the exit node} \\ \bigcup_{j \in succ(i)} c_{ij} & \text{otherwise} \end{cases} \tag{5.5}$$

where c_{ij} is defined as:

$$c_{ij} = \begin{cases} \{(i,j)\} & \text{if } FREQ_{ij} \leq cost(Cin_j) \\ Cin_j & \text{otherwise} \end{cases} \tag{5.6}$$

The main part of the equation for $Cout_i$ says that store s can be made fully anticipable on exit from basic block i if we pay the price of making it anticipable on each outgoing edge from i. The lowest cost of making it anticipable on an edge (i,j) is either the least cost of making s anticipable at entry to block j (i.e., the solution for Cin_j) or it is the cost of inserting s on the edge (i,j). The use of a *less-or-equal* comparison in the definition of c_{ij}, as opposed to a *less-than* comparison is deliberate, since it encourages insertions to occur at the earliest possible point and will therefore avoid unnecessary code motion.

An iterative approach for solving the above data-flow system is guaranteed to converge to a fix point. If we start by initializing all the Cin and $Cout$ sets to empty, except for the $Cout$ set of the exit node which should be initialized to $\bot$, then the total cost of the sets (as measured by the *cost* function) can only grow monotonically while the iteration proceeds. The sets themselves can both increase and decrease in cardinality, but the computed cost for a set can never decrease. Since there is an upper bound for each cost set and because the number of possibilities for an increment in cost is finite, convergence in a finite number of steps is assured.

5.5.3.2 Benefit analysis

The above section only gives the cost sets, both at entry and exit, associated to every basic block in the flow graph. The goal now is to obtain (a) a set of blocks that contain redundant instances of the considered store s, and (b) a set of edges where new instances of store s need to be inserted.

If a basic block i contains an instance of store s that is not subsequently read (i.e., $AVAIL_i = true$), then store s is a *candidate* for elimination. In this is the case, we have:

- The benefit to be derived from eliminating s from i would be $FREQ_i$ (i.e., the frequency of execution of basic block i).

- The set $Cout_i$, as said before, represents the cost of making s fully redundant at exit from basic block i.

```
Input:
    $G = (N, E)$                        # control-flow graph
    $\{Cout_i \mid i \in N\}$           # set of cost sets
    $\{FREQ_i \mid i \in N\}$           # set of execution frequencies
    $\{AVAIL_i \mid i \in N\}$          # set of availabilities (boolean)
Output:
    $Insert \subset E$                  # set of insertion edges
    $Redund \subset N$                  # set of redundancies (blocks)
Method:
    $Cand := \{i \mid i \in N \wedge AVAIL_i = true\}$
    $Insert := \emptyset$
    $Redund := \emptyset$
    while $\exists i \in N \mid FREQ_i > cost(Cout_i - Insert)$ do
        $Insert := Insert \cup Cout_i$
        $Redund := Redund \cup \{i\}$
        $Cand := Cand - \{i\}$
    end while
End Method
```

Figure 5.6: Benefit analysis for partial-SRE.

If we found that this execution frequency is greater than $cost(Cout_i)$, we would speed up the program by inserting s on all edges in the set $Cout_i$ and deleting s from i. Note that there may be another block j where $FREQ_j \leq cost(Cout_j)$, but $Cout_j$ contains some edges in common with $Cout_i$. If we have already decided to insert s on the edges in $Cout_i$ then the *additional* cost of making s fully redundant at exit from block j may be less than $FREQ_j$.

We apply the above algorithm, which is presented in Figure 5.6, to find as many opportunities as possible where instances of s can be profitably eliminated. A complete version of this analysis would be combinatorial in nature, which results in having exponential running time. This simple analysis will find quickly the major code motion opportunities in the flow graph.

5.5.3.3 Final cost-benefit equations

The last step of the analysis will check the benefit of removing every instance of store s considering register availability and cost (measured by execution latency) of instructions. First, we compute the benefit of removal as follows:

$$B \quad = \quad lat_{store} \times BB_{red}^{freq} \tag{5.7}$$

As we can see, the benefit will depend on the latency of the store to be removed weighted by its execution frequency. Usually, store instructions are assumed to have no latency, since they have no true dependencies with the following non-memory instructions. However, we assume in this analysis that the cost of executing a store is greater than the cost of executing a move operation (which is used for bypassing the value to be stored to the program points of new insertions). We make such assumption in order to exploit hidden benefits of removing stores, such as exposing dead code for elimination. The cost is then as follows:

$$C = lat_{move} \times BB_{red}^{freq} + lat_{store} \times \sum_{i=1}^{m} EDG_i^{freq} \tag{5.8}$$

The cost C will consider the cost of bypassing values (i.e., adding moves) and the cost of new insertion (i.e., adding stores). Bypassing the value to be stored needs of a previous analysis for computing register availability, followed by a phase for choosing the best register to be used as value container (see Section 5.5.1). This cost is a pessimistic upper bound, since the algorithm would avoid some of the move instructions. However, if the algorithm for choosing available registers fails, some opportunities for store removal may be lost.

Finally, after we compute the above equations, we apply SRE only if the benefit of eliminating the store instruction is greater than its cost:

$$SRE \Leftrightarrow C < B \tag{5.9}$$

The use of a *less-than* comparison in the definition instead of a *less-or-equal* is deliberate, since it tries to avoid unnecessary code motion.

5.6 More aggressive MRE techniques

Although the strategies seen so far are able to catch a high percentage of memory redundancy, they still fail in some cases. In this section we extend our redundancy elimination approaches with a couple of additional techniques, so that a much higher percentage of binary redundancy can be detected and eliminated.

5.6.1 A path-sensitive formulation for partial LRE

Figure 5.7 shows an example where, even considering partially redundant loads, our algorithms are not able to find out the existing redundancy between load instructions L_1 and L_2. The reason is that the default alias analysis we use for disambiguating memory references (see Section 2.2.3), like most analyses used in optimizing compilers, is unable to recognize memory dependencies in a path-sensitive way. We call *path-sensitive redundancy* the type of redundancy that exists only along some (but not all) execution paths leading to the redundant instruction. Clearly, a lot of redundancy is path sensitive.

To remove path-sensitive redundancy, unfortunately, the optimizer has to pay the *exponential* price of optimizing each path separately. The reason why analyzers avoid this situation

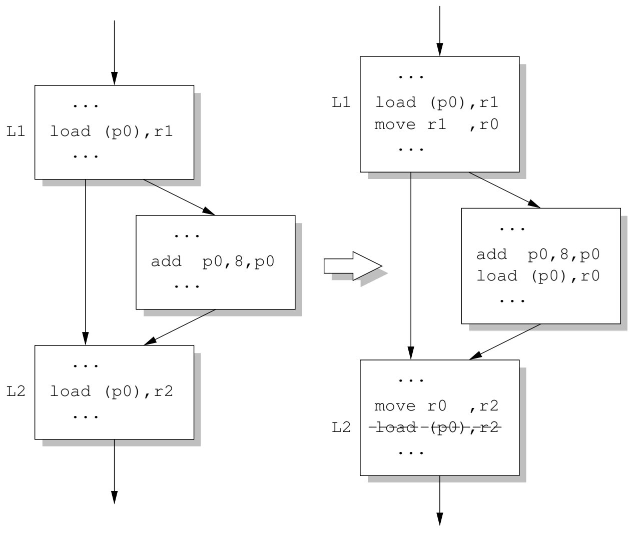


Figure 5.7: Elimination of a path-sensitive redundant load. A path-sensitive memory disambiguation mechanism is needed to detect the existing redundancy.

is that there is an exponential number of paths, even in a program with no loops [Bod99]. This is the reason why analyzers treat paths together, summarizing their results whenever paths meet, therefore diluting optimization opportunities. Unfortunately, paths explode exponentially not only in analysis, but also in program transformation, when we want to exploit the optimizable paths. As an example, another solution to enable optimization in Figure 5.7 might be to physically separate the considered paths via code duplication, but this duplication may also cause an exponential code growth, thus lowering performance.

Previous research (see Section 5.8) showed that existing MRE techniques are mainly based on path-insensitive information, which causes many MRE opportunities to be lost. In this section, we present a new technique for eliminating memory redundancies in a path-sensitive fashion. The key to our new proposal is to extend the alias analysis we proposed in Section 4.2.2 to recognize *path-sensitive memory redundancies*, and then use this information to guide a more accurate MRE algorithm. We will use path-sensitive information only when applying partial-LRE, since expectations are higher for load redundancy.

The final step consists in adapting our implementation of partial-LRE, which was presented in Section 5.5.1, by modifying the original Horspool and Ho's equations [HH97b] in order to deal with the path-sensitive disambiguation we presented in Chapter 4. Actually, the only change is related to the cost of making the candidate available on edge (i, j) of the flow graph:

$$c_{ij} = \begin{cases} \{(i,j)\} & \text{if } \overline{AVPATH_{ij}} \quad (*) \\ \{(i,j)\} & \text{if } FREQ_{ij} \leq cost(Cout_i) \\ Cout_i & \text{otherwise} \end{cases} \tag{5.10}$$

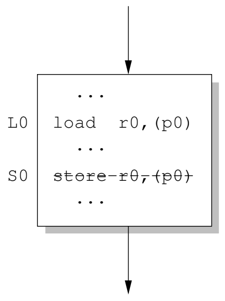


Figure 5.8: Elimination of a dead store.

From the new $(*)$ case, a new insertion in edge (i,j) is now required if the candidate load is not *available* on such edge. The new *availability on edge* property (i.e., $AVPATH_{ij}$) results form the application of path-sensitive disambiguation as presented in Definition 4.2.5 (see Section 4.2.2 in Chapter 4).

To the best of our knowledge, only Bodík and Anik [BA98, Bod99] described the problem of optimizing path-sensitive redundancies by using a new representation called *Value Name Graph (VNG)*. However, it is not obvious that using the VNG for optimizing executable code would scale up to problems of this size. As far as we know, the scheme for path-sensitive disambiguation developed here is the first attempt of exposing path-sensitive memory redundancies by simply extending the memory disambiguation algorithm as we did in Section 4.2.2.

5.6.2 Eliminating dead stores

Dead code elimination (see Section 2.3.1.2) removes instructions that can be proven to have no effect on the result of a computation [ASU86a]. This is an important optimization, not only because some programs contain dead code as originally written, but also because many of the other optimizations create dead code. For this reason, it is usually applied several times during an optimizing compilation.

As far as optimizing executable code is concerned, implementation of dead code elimination is often solely based on register liveness information, due to the fact that memory disambiguation is one of the weak points of object code modification (as we already discussed in Chapter 4). As a result, neither traditional dead code removal nor MRE approaches seen so far will detect useless store instructions writing into a memory cell the same value previously read from the same location, which has not been modified in the meantime. An example of *dead store* and its elimination is shown in Figure 5.8.

Dead store elimination (DSE) may be an important optimization at executable code level because other optimizations produce a lot of dead stores [Muc97m]. As an example, procedure inlining exposes useless stack management (specially store instructions), which can be eliminated. Combining MRE and dead code removal may also produce load/store pairs as

MRE	Distance	Description	Sections
Basic	Short	LRE and SRE within EBBs	5.4.1
Full		Full-LRE, Full-SRE	5.4.2
Partial	Long	Partial-LRE, Partial-SRE	5.5.1, 5.5.2
Complete		Path-sensitive LRE, Partial-SRE, DSE	5.6.1, 5.5.2, 5.6.2

Table 5.1: Description of the different MRE algorithms under evaluation.

shown in Figure 5.8. Even without such optimizations, the *potential* number of dead stores is interesting enough, as we already showed in Figure 5.1b, although the percentages presented did not distinguish between dead stores and "normal" computation.

Our approach for eliminating dead stores is based on simple heuristics by using *use-def* chains. We first look for store instructions (a) having a load operation as the definition instruction for its source register, and (b) no other store between them may be in conflict with the considered memory location. In such a case, the store may be safely removed. We also employ a basic liveness analysis for stack locations to eliminate useless stores to the stack.

5.7 Evaluation

In this section, we describe the process we have followed for evaluating the effectiveness of the MRE techniques proposed in this chapter.

We have implemented the MRE approaches presented in Table 5.1 (and described in the previous sections) on the Alto framework we described in Section 3.2.1. The algorithms have been integrated with the rest of optimizations carried out by Alto, such as constant/copy propagation, dead/unreachable code elimination, inlining, etc. This approach ensures not only maximizing the benefits coming from MRE itself, but also enhancing the effect of the rest of Alto optimizations.

The MRE integration has been performed within the optimization scheme presented in Section 3.2.1.1, by following the next guidelines:

- We include into the base optimizations the short-distance MRE on extended basic blocks (i.e., basic-MRE, see Table 5.1). The reason is that, since computing the data-flow equations and performing the redundancy searches for an EBB is relatively cost-effective, it can be applied several times during the optimization process.

- Within the one-time optimizations, we apply one of the long-distance MRE algorithms (see Table 5.1): (a) full-MRE, (b) partial-MRE, both performing LRE and SRE separately; or (c) complete-MRE. These are expensive optimizations and, therefore, we perform them only once. Since the formulation of our algorithms is intraprocedural,

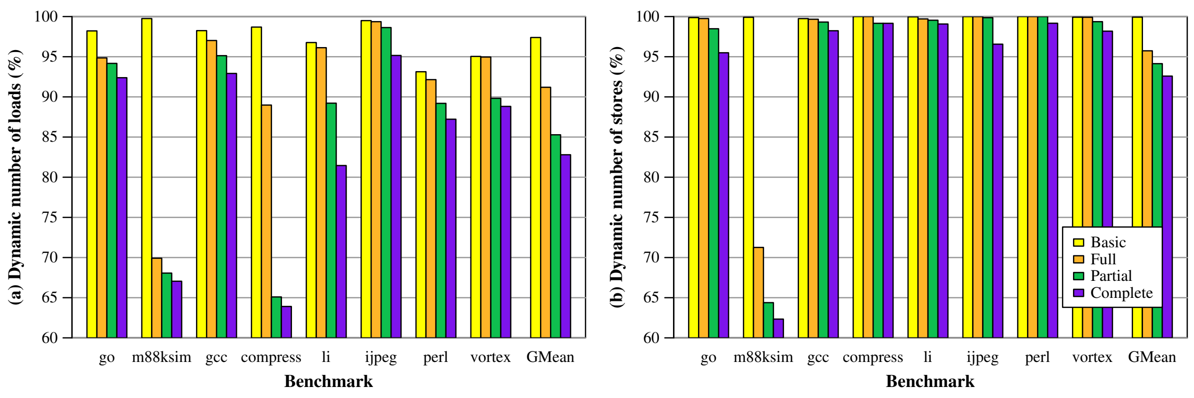


Figure 5.9: Effect of different MRE degrees in number of loads and stores at run time. The baseline (i.e., 100%) is Alto/Inline binaries without any MRE at all.

procedure inlining is previously performed by default so that our MRE proposals can have an interprocedural behavior.

To maximize the benefits of the different long-distance MRE optimizations, we apply the analysis phase of MRE to every function and keep a “per-function” list of all the candidates for removal sorted by net benefit. These benefits are recomputed every time that a candidate is removed, thus sorting the rest of candidates again. Then, the “most redundant” references are the ones that we try to remove first, when the chance of finding available registers inside the function is higher. Finally, to keep the running time of the corresponding MRE algorithm under control we use a ϕ value of 0.75 (see Section 3.2.1.3). The idea is to apply MRE only to the “hot references” in the program.

The benchmarks we have used for our experiments were presented in Section 3.3, and they were generated following the methodology described in Section 3.4. For our experiments, we have chosen the Alto/Inline binaries as a baseline (that is, the most optimized binaries we have), whose characteristics and generation procedure were described in Section 3.3.1.

5.7.1 Reduction in number of dynamic references

We start evaluating the effectiveness of the MRE algorithms under study by comparing the number of dynamic loads and stores executed with respect to the program baseline. To get these results, we ran our benchmark suite using the corresponding simulation inputs (see Table 3.2 in Section 3.3) on top of the `sim-profile` simulator of the SimpleScalar toolset [ALE02].

The two graphs in Figure 5.9 present the reduction in number of dynamic loads and stores respectively for each benchmark. As it can be seen, all programs do show improvements typically around 5–20% for loads and 1–5% for stores, with some better cases such as `m88ksim`

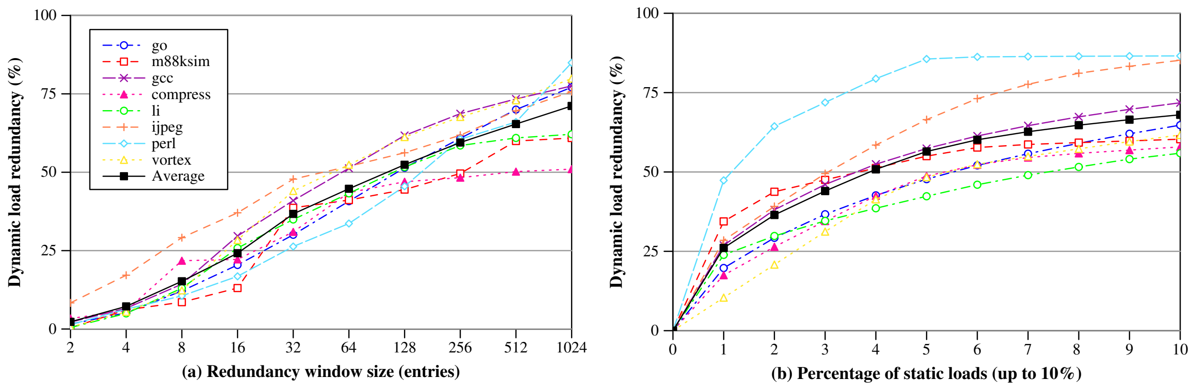


Figure 5.10: Dynamic amount of load redundancy after complete-MRE: (a) vs. redundancy window size, and (b) vs. percentage of static loads, for a 1024-entry redundancy window.

and `compress`. Figure 5.9a also shows that working only on EBBs is not enough to catch the load redundancy we presented in Section 5.2 (except maybe for `perl` and `vortex`), while targeting LRE to catch distant redundancy yields significant improvements for every benchmark. On the other hand, the results in Figure 5.9b show that reduction in dynamic stores is only important when DSE is applied, due to the fact that store redundancy is much lower than load redundancy, as we showed in Section 5.2. Program `m88ksim` is, however, an interesting exception. The reason is that, as LRE is performing so well removing redundant loads, a high percentage of stores become now redundant. This fact indicates that removing loads is crucial for removing stores, since it exposes "hidden" store redundancy.

The results we have obtained are consistent with those observed in Tables 2 and 3 in Lo *et al.* [LCK+98], although we achieve rather less benefits on load removal. We believe there are two main reasons for that: (a) we do not have the advantage of high-quality alias analysis as they do, and (b) we are working on final machine code while they were measuring reduction in dynamic loads *before* register allocation and code generation.

Finally, Figure 5.10 presents two graphs showing the percentage of load redundancy remaining after applying complete-MRE[6], which can be compared to the results presented in Figure 5.1. As we can see, MRE achieves around a 20% reduction in dynamic load redundancy, both varying the redundancy window and considering the percentage of static loads.

5.7.2 MRE breakdown of static references

In order to better understand the above results, it is worth looking at the internals of our algorithms. Table 5.2 and Table 5.3 break down the opportunities for load and store removal respectively, for each of the MRE algorithms under evaluation. For each algorithm, three classes are presented: *Can*, *Ben* and *Rem*. Class "Can" indicates the number of loads/stores

[6]Results for store redundancy are not presented, since reduction levels are not significant in this case.

Benchmark	**Brk**	**Basic**	**Full**	**Partial**	**Complete**
`099.go`	Can	547 (1.00)	712 (1.30)	798 (1.46)	1043 (1.91)
	Ben	457 (1.00)	587 (1.28)	610 (1.33)	823 (1.80)
	Rem	448 (1.00)	540 (1.21)	562 (1.25)	781 (1.74)
`124.m88ksim`	Can	606 (1.00)	731 (1.21)	814 (1.34)	870 (1.44)
	Ben	517 (1.00)	621 (1.20)	677 (1.31)	732 (1.42)
	Rem	499 (1.00)	603 (1.21)	659 (1.32)	714 (1.43)
`126.gcc`	Can	2802 (1.00)	2868 (1.02)	3040 (1.08)	3446 (1.23)
	Ben	1893 (1.00)	1952 (1.03)	2029 (1.07)	2365 (1.25)
	Rem	1840 (1.00)	1897 (1.03)	1974 (1.07)	2310 (1.26)
`129.compress`	Can	268 (1.00)	291 (1.09)	332 (1.24)	344 (1.28)
	Ben	226 (1.00)	247 (1.09)	291 (1.29)	302 (1.34)
	Rem	217 (1.00)	238 (1.10)	273 (1.26)	284 (1.31)
`130.li`	Can	1192 (1.00)	1219 (1.02)	1783 (1.50)	2310 (1.94)
	Ben	672 (1.00)	811 (1.21)	1279 (1.90)	1721 (2.56)
	Rem	654 (1.00)	793 (1.21)	1261 (1.93)	1703 (2.60)
`132.ijpeg`	Can	434 (1.00)	446 (1.03)	484 (1.12)	568 (1.31)
	Ben	388 (1.00)	400 (1.03)	431 (1.11)	512 (1.32)
	Rem	379 (1.00)	388 (1.02)	419 (1.11)	474 (1.25)
`134.perl`	Can	1451 (1.00)	1471 (1.01)	1521 (1.05)	1550 (1.07)
	Ben	1259 (1.00)	1276 (1.01)	1313 (1.04)	1337 (1.06)
	Rem	1230 (1.00)	1247 (1.01)	1284 (1.04)	1308 (1.06)
`147.vortex`	Can	1992 (1.00)	1998 (1.00)	2252 (1.13)	2336 (1.17)
	Ben	1857 (1.00)	1863 (1.00)	2105 (1.13)	2184 (1.18)
	Rem	1818 (1.00)	1824 (1.00)	2066 (1.14)	2145 (1.18)
Total	Can	9292 (1.00)	9736 (1.05)	11024 (1.19)	12467 (1.34)
	Ben	7269 (1.00)	7757 (1.07)	8735 (1.20)	9976 (1.37)
	Rem	7085 (1.00)	7530 (1.06)	8498 (1.20)	9719 (1.37)

Table 5.2: Static MRE on loads for our benchmark suite. Categories stand for *Can*: number of candidates, *Ben*: opportunities after applying cost-benefit analysis, and *Rem*: loads actually removed. Relative numbers with respect to basic-MRE are also presented in parenthesis.

considered by the algorithm as candidates for removal. Note that this number is computed *after* memory disambiguation has determined that there are no conflicting aliases which prevent MRE. The second class, “Ben”, are the number of candidates remaining *after* applying our cost-benefit analyses. For example, the large drop in Table 5.2 for `gcc` indicates that the costs of removing those loads out-weight the expected benefits. Finally, class “Rem” indicates the number of static loads/stores actually removed. The differences between class “Ben” and “Rem” are attributed to the lack of registers when bypassing the candidate values. The table also presents relative numbers with respect to basic-MRE (i.e., the first column), to give an numeric estimator of the improvement of every MRE algorithm.

As we can see, the lack of registers is directly responsible for only a minority of the “lost opportunities”, thus suggesting that register pressure is far from being the main problem in applying MRE. The effect is even greater in Table 5.3 because register availability is less

Benchmark	**Brk**	**Basic**	**Full**	**Partial**	**Complete**
`099.go`	Can	27 (1.00)	28 (1.04)	40 (1.48)	389 (14.41)
	Ben	27 (1.00)	28 (1.04)	40 (1.48)	389 (14.41)
	Rem	27 (1.00)	28 (1.04)	40 (1.48)	389 (14.41)
`124.m88ksim`	Can	74 (1.00)	89 (1.20)	110 (1.49)	197 (2.66)
	Ben	74 (1.00)	89 (1.20)	110 (1.49)	197 (2.66)
	Rem	74 (1.00)	89 (1.20)	110 (1.49)	197 (2.66)
`126.gcc`	Can	42 (1.00)	44 (1.05)	46 (1.10)	531 (12.64)
	Ben	42 (1.00)	44 (1.05)	46 (1.10)	531 (12.64)
	Rem	42 (1.00)	44 (1.05)	46 (1.10)	531 (12.64)
`129.compress`	Can	5 (1.00)	5 (1.00)	9 (1.80)	51 (10.20)
	Ben	5 (1.00)	5 (1.00)	9 (1.80)	51 (10.20)
	Rem	5 (1.00)	5 (1.00)	9 (1.80)	51 (10.20)
`130.li`	Can	74 (1.00)	91 (1.23)	134 (1.81)	238 (3.22)
	Ben	74 (1.00)	91 (1.23)	107 (1.45)	211 (2.85)
	Rem	74 (1.00)	91 (1.23)	107 (1.45)	211 (2.85)
`132.ijpeg`	Can	20 (1.00)	20 (1.00)	23 (1.15)	109 (5.45)
	Ben	20 (1.00)	20 (1.00)	23 (1.15)	109 (5.45)
	Rem	20 (1.00)	20 (1.00)	23 (1.15)	109 (5.45)
`134.perl`	Can	56 (1.00)	56 (1.00)	56 (1.00)	266 (4.75)
	Ben	56 (1.00)	56 (1.00)	56 (1.00)	266 (4.75)
	Rem	56 (1.00)	56 (1.00)	56 (1.00)	266 (4.75)
`147.vortex`	Can	77 (1.00)	78 (1.01)	89 (1.16)	237 (3.08)
	Ben	77 (1.00)	78 (1.01)	89 (1.16)	237 (3.08)
	Rem	77 (1.00)	78 (1.01)	89 (1.16)	237 (3.08)
Total	Can	375 (1.00)	411 (1.10)	507 (1.35)	2018 (5.38)
	Ben	375 (1.00)	411 (1.10)	480 (1.28)	1991 (5.31)
	Rem	375 (1.00)	411 (1.10)	480 (1.28)	1991 (5.31)

Table 5.3: Static MRE on stores for our benchmark suite. Categories stand for *Can*: number of candidates, *Ben*: opportunities after applying cost-benefit analysis, and *Rem*: stores actually removed. Relative numbers with respect to basic-MRE are also presented in parenthesis.

important when eliminating store instructions. This is the reason why there is no drop for stores on class “Rem”. As far as load instructions are concerned, the largest drop corresponds to class “Ben”, where our cost-benefit analysis discards many opportunities for load removal due to several reasons. First, as we note in Section 5.4.1, not every redundant load is profitable for removal. Besides, as the short-distance LRE is carried out several times in the process, some of these non-profitable loads that do not change its cost-benefit status are repeatedly counted. Another reason relies on the fact that, since our cost equations are conservative, they always assume that move instructions will be inserted when needed, regardless of whether a later copy propagator will be able to remove them or not.

In general, Table 5.2 and Table 5.3 show that every MRE algorithm is important when removing loads, while for store instructions the opportunities come from removing dead stores. The conclusion to be drawn is that, probably, we should focus future work on improving the

Category	Description	Sections
Base	no MRE, no inlining (i.e., Alto/Base)	3.3.1
Inline	no MRE, inlining (i.e., Alto/Inline)	3.3.1
noInline	Complete-MRE, no inlining	5.7.3
noInline'	Complete-MRE, no inlining (without calling overhead)	5.7.3
Complete	Complete-MRE, inlining	5.7

Table 5.4: Description of the binaries obtained with/without applying MRE and inlining. The baseline will be *Inline* binaries (i.e., Alto/Inline binaries without any MRE at all).

number of candidates that our MRE algorithm targets (i.e., the "Can" class), to fully obtain the potential of the MRE optimizations.

5.7.3 Effects of procedure inlining on MRE

Since the formulation of our MRE algorithms is intraprocedural, procedure inlining is previously performed by default so that our MRE proposals can have an interprocedural behavior and obtain benefit from it. However, it would be interesting to measure how procedure inlining will affect our analyses and MRE optimizations in "hiding" memory redundancy to be discovered and eliminated. This does not only include interprocedural redundancy, but also intraprocedural redundancy discovered after obtaining interprocedural information (e.g., when two function arguments are pointers which point to exactly the same location).

Table 5.4 presents the different benchmark sets we use for evaluating the effect of inlining on MRE, which is done by measuring the reduction in number of dynamic loads and stores presented in the two graphs of Figure 5.11, respectively. These binaries result from enabling/disabling MRE and procedure inlining in Alto when optimizing our benchmark suite. The *Base* and *Inline* categories correspond to the Alto/Base and Alto/Inline sets we already evaluated in Section 3.3.1. Actually, data in Figure 5.11 for these sets was already presented in Figure 3.4. Category *noInline* means that complete-MRE is performed with no previous procedure inlining at all. However, as we will explain next, inlining by itself is able to remove some loads and stores, which makes difficult to compare references between programs with and without inlining. Therefore, we introduce the *noInline'* category, which is the same as *noInline*, but we manually subtract from the final number of loads and stores those that were removed thanks to inlining in the baseline programs, without any MRE at all. We perform such "trick" since it would be very unfair to compare removed references across different baselines. Finally, the *Complete* category corresponds to the complete-MRE already presented in Figure 5.9 (including procedure inlining), which has been added to the figure just for comparison purposes[7].

[7]To be consistent with previous sections, we maintain the Alto/Inline binaries as the baseline in Figure 5.11.

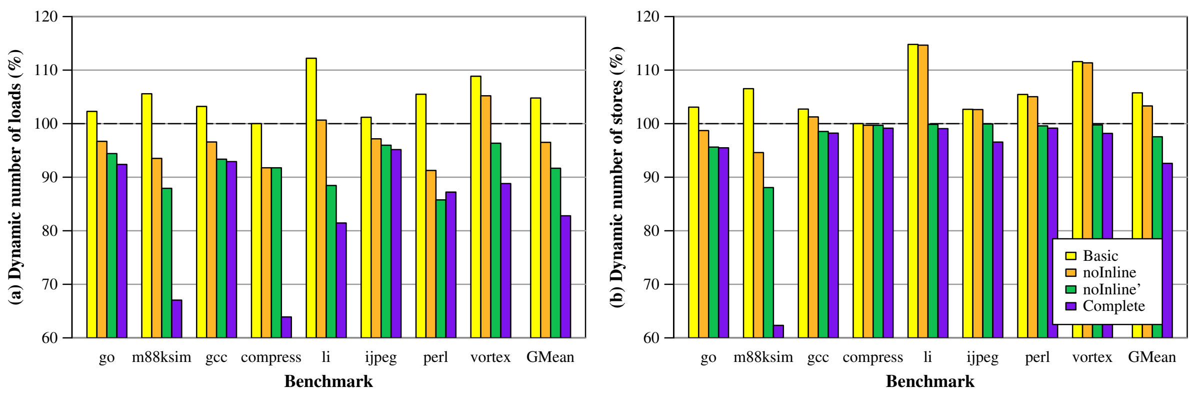


Figure 5.11: Effect of procedure inlining on the MRE algorithms, measured as number of loads and stores at run time. The baseline is Alto/Inline binaries without any MRE at all.

From the results presented in Figure 5.11 we can observe first that programs on the *Base* benchmark set execute more loads and stores than the *Inline* baseline (i.e., relative numbers are higher than 100%). The reason is that, when inlining is applied, a significant percentage of calling convention overhead can be removed, which includes saving temporary registers in the procedure stack frame. As a result, Alto/Inline binaries already include some form of useless load and store elimination. The fact that this removal (i.e., elimination of unnecessary references due to calling convention overhead when applying inlining) is already included as a baseline in all our experiments makes our results even more valuable (i.e., the ones presented in Figure 5.9). A second thing to note is that *noInline* binaries achieve in general better load reductions than the inlined baseline programs. However, they are really far away from the reductions obtained when applying complete-MRE, although this distance is reduced when considering the "estimated" *noInline'* programs. This result seems to indicate that there exists a significant amount of interprocedural memory redundancy in these programs, which points us out that either procedure inlining or an intraprocedural MRE formulation is needed for eliminating redundant references.

5.7.4 Compilation time

Related to how procedure inlining affects MRE, we can see in Figure 5.12 the relative compilation time of Alto when applying the MRE algorithms as presented in Table 5.1, with respect to our baseline. We also include in the figure the data corresponding to the application of complete-MRE without any procedure inlining at all. First, Figure 5.12 shows that global MRE schemes do not come without a cost, specially on those programs where inlining was widely applied and the percentage of *hot* basic blocks is significant. In general, procedure inlining dramatically increases MRE compilation time, while for the non-inlined binaries (i.e., the right bars) the compile-time overhead of MRE is more moderate. Another interesting ob-

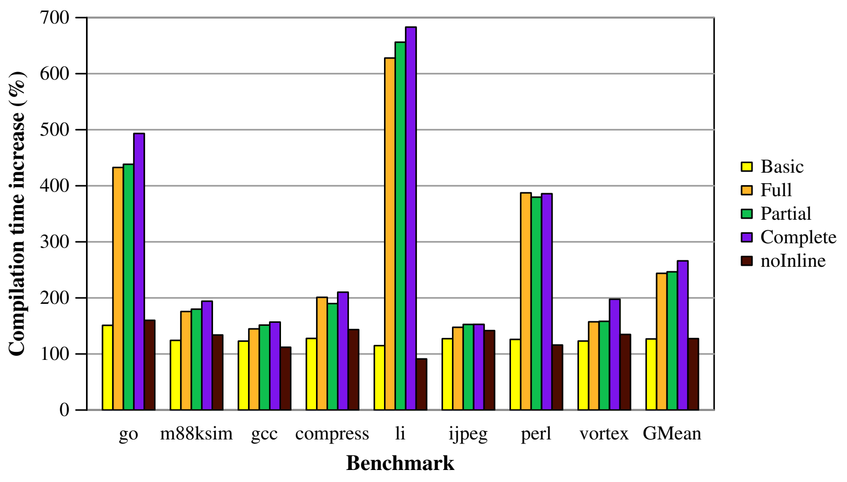


Figure 5.12: Effect of applying MRE in Alto compilation time, for the SPECint95 programs. The baseline is Alto/Inline binaries without any MRE at all.

servation is that, even though the path-sensitive disambiguation included in complete-MRE seemed to be very costly (see Section 4.2.2), the used heuristics prove to be very effective in not increasing compilation time too much with respect the partial-MRE algorithm.

5.7.5 Speed up using MRE

Counting the number of removed references is certainly of interest to understand the effectiveness of each algorithm. However, the final measure of interest is whether execution time is reduced or not. Figure 5.13 presents the relative execution time of our benchmark suite after the different MRE algorithms, using the execution inputs (see Table 3.2 in Section 3.3). These results were recorded by running the benchmark suite in our target platform, as we described in Section 3.1.3.

From the results presented in Figure 5.13 we can see that, of course, since memory references are only a fraction of all instructions executed in a program, reduction in execution time is smaller than the corresponding reduction in number of dynamic references. Thus, looking at the geometric mean, the 18% (8%) reduction in dynamic loads (stores) observed in Figure 5.9 only translates into a 10% reduction in execution time. Overall, however, the observed decrease for all programs shows that (a) we have removed some references that indeed were on the program's critical path and, therefore, contributed heavily to final execution time, and (b) other optimizations carried out by the optimizer do take advantage of the memory instructions removed, thus resulting in more efficient code.

By looking at every benchmark, we can see that MRE yields important reductions for the majority of the programs, while for programs `gcc` and `ijpeg` the obtained reductions were relatively modest. Nevertheless, the cost in compilation time on these cases was not very

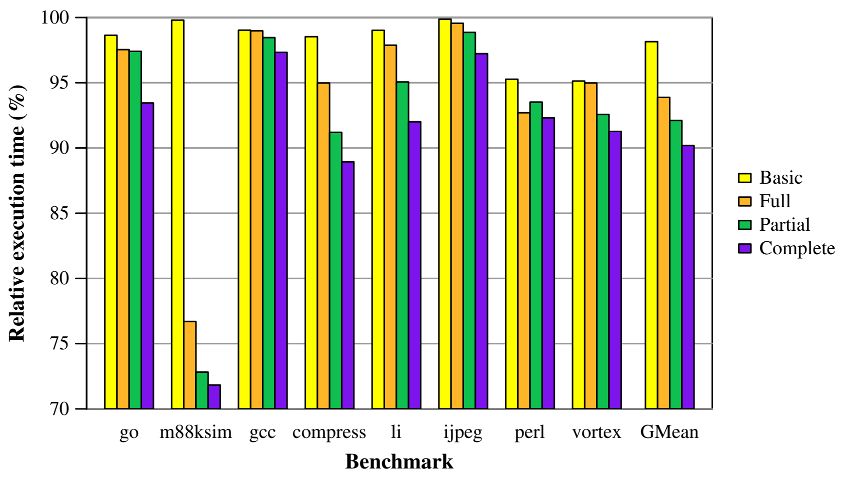


Figure 5.13: Effect of different MRE degrees in actual execution time. The baseline is Alto/Inline binaries without any MRE at all.

high either. Finally, note that every MRE algorithm seems to be important for reducing the final execution time, and that choosing which MRE method has greatest benefits is strongly dependent on which benchmark it is being applied.

5.7.6 Microarchitecture impact of using MRE

It would be interesting to get an accurate measure of the differences at the processor-core level of applying the MRE algorithms. To this end, we have simulated our benchmark suite on the `sim-alpha` simulator we introduced in Section 3.2.2 [DBK01], using the corresponding simulation inputs (see Table 3.2 in Section 3.3). This out-of-order simulator faithfully models a Compaq/Alpha 21264 configuration that matches our target environment presented in Section 3.1.1.

Three interesting points can be pointed out by analyzing the low-level internal statistics produced by the simulator:

- Figure 5.14 presents the dynamic reduction in number of *replay traps* for each benchmark with respect to the original baseline[8]. As we can observe, the number of replay traps is drastically reduced (up to 70%), which heavily contributes to reduce final execution time. However, there are few cases where replay traps are introduced, being program `perl` after applying partial-MRE the worst case. We believe this is the reason why this program showed a small slowdown in Figure 5.13.

[8]A *memory replay trap* occurs in the Alpha 21264 when a load is found to have issued to memory out of order with respect to an older memory operation that overlaps [Com99b]. If this is the case, the instruction is aborted (along with all newer instructions) and restarted from the fetch stage of the pipeline.

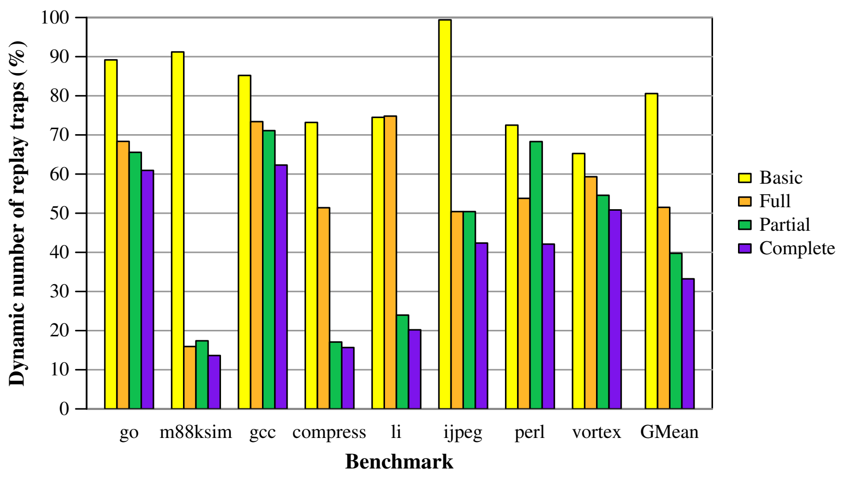


Figure 5.14: Effect of different MRE algorithms in number of replay-traps. The baseline is Alto/Inline binaries without any MRE at all.

- Second, miss-ratio at L1 data-cache was slightly higher when applying MRE, due to the fact that, as we are removing memory redundancies, less "hit" memory accesses are requested to the memory hierarchy. Difference, nonetheless, is under 1% for all programs.
- Another difference is in the number of instructions per cycle (IPC) reached when applying MRE, which was also slightly higher. The reason for this effect is that MRE is breaking memory dependencies by turning them into register dependencies, which are easier solved by the hardware, and thus increase IPC.

Finally, the relative execution times we obtained from these simulations, which will be also presented in next section, are quite consistent with the results presented in Figure 5.13. Small differences are attributed to simulation inaccuracy, as shown in [BS98].

5.7.7 Effects of load latency

Another interesting measure to gauge the importance of the MRE transformation is to see what will happen in the future, as L1-cache latency continues to increase. Current CPUs are typically at a 2-cycle or 3-cycle load latency and the trend is towards hyper-pipelining and, therefore, longer latencies. In this section we re-simulated all the benchmarks on the simulator we used in the previous section, changing the L1-cache latency from its value of 3 cycles up to 4 and 5 cycles. Furthermore, for each experiment we also re-compiled every benchmark, since all our cost-benefit analyses are dependent on the latency of the loads. At larger latencies, it is more likely that the cost-benefit equations tend to favor the substitution of a load by one or more "move" instructions.

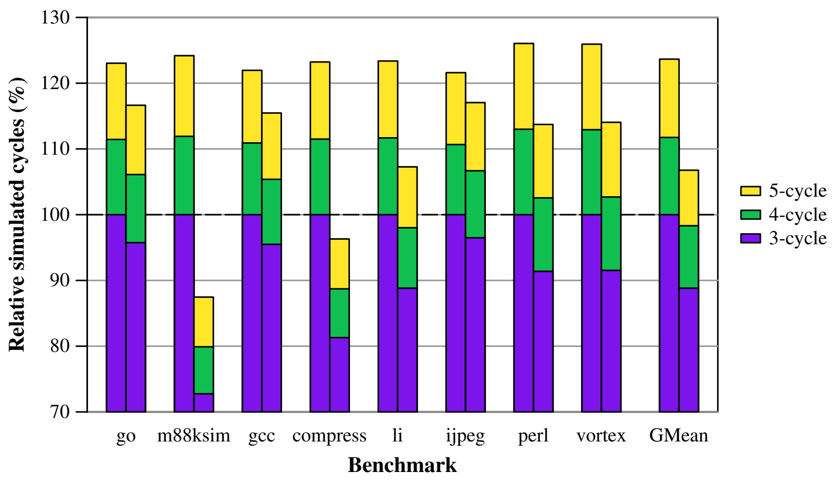


Figure 5.15: Effects of load latency, from 3- to 5-cycle hit latency. The left bars correspond to Alto/Inline binaries without any MRE at all, being the baseline the 3-cycle latency execution. The right bars correspond to binaries obtained after applying complete-MRE.

The results can be seen in Figure 5.15, where we present data for different latencies with/without applying complete-MRE in our benchmark suite. The baseline is Alto/Inline binaries without any MRE at all, considering a 3-cycle latency for loads. The first thing to observe is that, as we mentioned in the previous section, relative execution times obtained from these simulations for the 3-cycle latency is quite consistent. Another obvious observation is that the longer the latency the worse the execution time of all programs. However, as latency increases, the importance of performing MRE also grows. Consider, for example, the cases of `m88ksim` and `compress`. After applying complete-MRE, the execution time at a 5-cycle latency is *even better* than the original execution time using a 3-cycle latency. In general, our results show that if we apply MRE, we can increase the latency of the data cache in one cycle without any performance degradation at all.

5.8 Related work

While a number of systems have been described for optimization of executable code (see Section 2.4), to the best of our knowledge, any elimination of redundant memory operations carried out by these systems is limited to fairly simple removal.

Memory redundancy elimination can be seen as a particular case of *Partial Redundancy Elimination* (PRE), where the expressions to be considered for removing are only memory operations. PRE [MR79, KRS94a, CCK$^+$97] is a well-known scalar optimization that subsumes various *ad hoc* code motion optimizations (such as common subexpression elimination and loop invariant code motion) by attempting to remove redundancies that occur only on some control-flow paths of a program. Horspool and Ho [HH97b] described a speculative formulation of PRE based on a cost-benefit of the flow graph, by using edge profiles (our current implementation of distant MRE optimizations is based on their equations). This approach has been generalized to provide optimal solutions both for time [CX03] and space [SHK04]. Gupta, Berson and Fang [GBF98] extended this PRE algorithm by using path profiles. Finally, Bodík, Gupta and Soffa [BGS98] developed a profile driven PRE approach using path profiles and control-flow restructuring, which is complete. However, as they replicate regions of code when needed, some code growth also results.

Register Promotion allows scalar values to be allocated to registers for regions or their lifetime, where the compiler can prove that there are no aliases for the value. Promotion carries out elimination of both redundant loads and stores [CCK90]. Cooper and Lu [CL97] examined promotion over loop regions. Their results indicate that the main benefit of promotion comes from removing store operations. Lo *et al.* [LCK$^+$98] use a variant of SSA-PRE to remove unnecessary loads and stores over any program region. However, they do not consider the effect of spilling because they simulate with an infinite symbolic register set before register allocation. Both works only counted the improvement compared to the total number memory instructions. Postiff, Greene and Mudge [PGM00b] presented a register promotion algorithm at link time, although their algorithm does not use any PRE approach at all. They also present numbers for long register files, but the gain in this case comes from several *ad hoc* techniques for promoting globals and constants into a dedicated subset of the register file.

The problem of path-sensitive redundancies has been described by Bodík and Anik [BA98, Bod99]. They propose a new representation called *Value Name Graph (VNG)* to be used for general path-sensitive optimizations. However, it is not obvious that using the VNG for optimizing large programs would scale up to problems of this size. By using the VNG, Bodík, Gupta and Soffa [BGS99] developed a load redundancy analysis and design a method for evaluating its precision, although their paper is only focused in the analysis and they do not perform any elimination of redundant loads at all.

5.9 Conclusions

In this chapter we have shown that, even although the compiler may have optimized a program aggressively, a significant number of redundant memory references appear in the final executable file. This *memory redundancy* does not only appear due to how programmers write source code, but also due to limitations in the compilation model of traditional compilers. To address these issues, we have presented a set of different algorithms to be applied in the context of binary or link-time optimizers, which are targeted to discover memory operations that are redundant and can be safely removed in order to speed up a program. We call this optimization *Memory Redundancy Elimination (MRE)*.

First, we have quantify how much memory redundancy is present in executable programs, and shown that around 75% of load references (10% of store references) can be considered redundant because they access memory locations that have already been referenced within a short dynamic distance (less than 256 references away in our dynamic window experiments). Then, we presented several profile-based MRE algorithms targeted at optimizing away these redundancies:

- The first MRE algorithm is targeted at catching short-distance redundancy at a very low compile-time cost, by looking at redundancy within extended basic blocks. this algorithm is able to remove less than 4% of all loads (1% of all stores) and, therefore, yields reductions below 3% in execution time. The results seem to indicate that an extended basic block is too small of a region to catch the redundancy measured in our redundancy experiments.
- The second algorithm presented, MRE for removing fully/partially redundant references over regions of arbitrary control-flow complexity, yields an average reduction of a 15% in dynamic loads (6% in dynamic stores). This results in a reduction over the baseline of 8% in execution time, although its application may increase significantly compilation time on those programs where procedure inlining has been aggressively applied.
- Finally, complete-MRE (the third algorithm discussed) couples the application of a new technique for eliminating redundant loads in a path-sensitive fashion, with a simple set of heuristics for removing dead stores. This combination performs a small increase in the number of dynamic references removed (18% for loads, 8% for stores). Despite this small increase, complete-MRE does detect some of the critical references and thus reduces execution time up to a 10%. Besides, it carries almost no optimization-time overhead with respect to a partial-MRE solution.

We have provided exhaustive experimental results in order to measure the effectiveness of the MRE algorithms under study. Thus, we ran the resulting optimized programs on top of a cycle-by-cycle simulator to give accurate measurements of the impact of applying MRE at the processor-core level. We also tested our optimizations assuming different cache latencies, and showed that, if latencies continue to grow, the load redundancy elimination will become more important. Overall, our results show that a significant amount of memory redundancy can indeed be eliminated, which translates into important reductions in execution time.

Chapter 6

Conditional branch redundancy elimination

In this chapter we continue revisiting the problem of redundancy in binary programs, by reasoning about the other big source of binary redundancy. We discuss this time the discovery and elimination of redundant conditional branches in the context of a link-time optimizer, an optimization that we call Conditional Branch Redundancy Elimination (CBRE)*. First, we motivate this work by measuring a potential upper bound on how much conditional branch redundancy is present in executable programs. We then propose several CBRE algorithms targeted at optimizing away these redundancies. Finally, we provide experimental results showing the effects of applying CBRE, including accurate measurements of the impact in code growth.*

6.1 Introduction

Branch predictors [Smi81, YP92, CG94, LCM97] are the best known hardware proposals for handling the problem of control dependencies in today's superscalar processors. As we pointed out in Section 1.1.1, one of the reasons why branch predictors are effective is because they exploit the *dynamic correlation* among branches. That is, they exploit the *dynamic conditional branch redundancy* existing in programs. Furthermore, recent research in branch prediction [Kra94, Pat95, YGS95, SLM96] and elimination of conditional branches [MW95, BGS97] has reported the existence of significant amounts of such branch correlation, presenting opportunities for optimizations when these branches are found to be *statically correlated* or *redundant*. A conditional branch is redundant along a path if the branch outcome can be determined along that path at compile time, from prior statements or branch outcomes [Bod99]. Figure 6.1 illustrates a C code example of redundant conditional branch, since condition `x == 1` is known to be false when condition `x > 0` evaluates false. The branch can be then safely removed by an appropriate program transformation.

Unnecessary conditional branches appear in a binary due to a variety of reasons: the compiler may not have the opportunity to perform intraprocedural constant propagation, thus executing branches on values that will be constant for sure; or maybe a branch is continuously evaluated from a value loaded from memory because the compiler was unable to resolve aliasing adequately. The elimination of these branches produces important benefits that explain why this can be an important optimization, such as improving hardware branch prediction or enhancing instruction scheduling and software pipelining.

We propose in this chapter an optimization to be applied in the context of binary or link-time optimizers. We discuss the discovery and elimination of redundant conditional branches, an optimization that we call *Conditional Branch Redundancy Elimination (CBRE)*. First, we quantify how much conditional branch redundancy is present in executable programs at run time, and show that a high percentage of conditional branches can be considered redundant because their outcomes can be determined from a previous small dynamic instruction stream. We also discuss how important memory disambiguation is in order to catch conditional branch redundancy. Then, we present several CBRE algorithms targeted at optimizing away these redundancies:

- A basic CBRE algorithm limited to extended basic blocks.
- A general algorithm that works over regions of arbitrary control-flow complexity, typically over functions.
- A profile-guided extension of the above algorithms targeted at eliminating redundant conditional branches in a path-sensitive fashion, where *control-flow graph restructuring* is required.

```
                 //  $x \in (-\infty, +\infty)$
if (x > 0)       //  $c_1 : x > 0\,?$
{
  ...
}
                 //  $x_{c_1} \in [1, +\infty)$,   $x_{\overline{c_1}} \in (-\infty, 0]$
if (x == 1)      //  $c_2 : x = 1\,?$
{
  ...
}
```

Figure 6.1: Example of redundant conditional branch, with value ranges of involved variables.

We also provide some experimental results in order to evaluate the effectiveness of the CBRE algorithms under study, including accurate measurements of the impact in code growth. Our results show that an important amount of conditional branch redundancy can indeed be eliminated, which translates into sensitive reductions in execution time.

6.2 Dynamic conditional branch redundancy

Before presenting our algorithms for removing redundant conditional branches, we motivate our work by measuring a potential upper bound on how much conditional branch redundancy could be removed from a program. To this end, we first introduce the analysis used for detecting conditional branch correlation.

6.2.1 Detecting branch correlation

Figure 6.1 illustrates a C code example of redundant conditional branch, since condition c_2 (i.e., `x == 1`) is known to be false when condition c_1 (i.e., `x > 0`) evaluates false. In this example we can easily identify a source of correlation from branch c_1 to c_2, but branch correlation may also come from constant assignments, register copies, common subexpressions, etc. Therefore, we need some form of symbolic-back substitution to capture branch correlation by propagating information about value ranges [Pat95, SBA00, BGSW00, MRS$^+$01, CGS04], as Figure 6.1 shows.

To perform such task, we use a *Value Range Propagation (VRP)* algorithm which is a simplified version of the analysis presented by Patterson [Pat95]. In our implementation, which is targeted at the executable code level, value ranges are defined by numeric lower and upper bounds, a data type, and some modifiers for the given range (e.g., complement, stride, etc.), but they could have been also symbolic in nature. Range operations include common computation needed for propagating ranges, such as union, intersection, subset, etc. The idea is to statically "build" the *computation tree* of the given branch (i.e., the branch that we want to remove) and every previous branch within the EBB. The computation tree or *slice* is built by using a scheme based on *register use-def chains* (see Section 2.2.2) that provide, for each use of a register, a pointer to its definition. Then, value ranges are propagated back and forth for every involved register, so that only the relevant data-flow information is computed [ABD+02, DLS02]. Correlation can be determined by simply consulting whether the final range of the branch source register is subsumed by the branch condition.

Figure 6.1 also illustrates how this algorithm works. Initially, value range of variable x is undetermined, which is denoted by the range $(-\infty, +\infty)$. Since variable x in involved in condition c_1, its value range before condition c_2 is promoted to one of the value ranges presented in the figure (i.e., either x_{c_1} or $x_{\overline{c_1}}$), depending on whether condition c_1 succeeds. It is straightforward to see that condition c_2 is known to be false when condition c_1 evaluates false, since condition c_2 is never accomplished when value range $x_{\overline{c_1}}$ holds. This analysis based on values ranges is general enough to be used for detecting both dynamic (i.e., at analysis time) and static (i.e., at compile time) branch correlation.

6.2.2 Measuring conditional branch redundancy

In this section we measure a potential upper bound on how much conditional branch redundancy is present in a program. To achieve this goal, we run on top of the SimpleScalar `sim-profile` simulator our original benchmark set with the corresponding simulation inputs (see Section 3.2.2 and Section 3.3, respectively) to capture every dynamic conditional branch instruction. Dynamic conditional branch redundancy is then measured by recording the most recent n conditional branches into a *redundancy window*. This window is a simple FIFO queue, where new branches coming into it displace the oldest ones stored in the window. Our goal is to measure how often the outcome of a conditional branch is already known at compile time, based on the previous dynamic information "stored" in the redundancy window, and also to quantify the typical distance (in conditional branch instructions) between such redundancies.

A dynamic instance of a conditional branch is considered redundant if its outcome can be determined from the *correlation information* that still remains in the redundancy window. If this is the case, the branch is said to have *dynamic correlation* from its prior instruction stream, including the outcome of other branches. Our analysis for discovering branch correlation uses the *Value Range Propagation (VRP)* algorithm we presented in Section 6.2.1. The idea is to dynamically "build" the computation tree or *slice* of every branch in the window, then propagating value ranges back and forth for the output of every instruction within the slice[1].

[1]The ideal analysis would be to have a non-linear system $\mathcal{S}$ of n inequations (i.e., the computation slice of every conditional branch in the window). Then, for every incoming branch, $\mathcal{S}$ should be solved in order to verify whether the system subsumes the condition of the incoming branch. Unfortunately, solving non-linear inequation systems is an infeasible problem in practice.

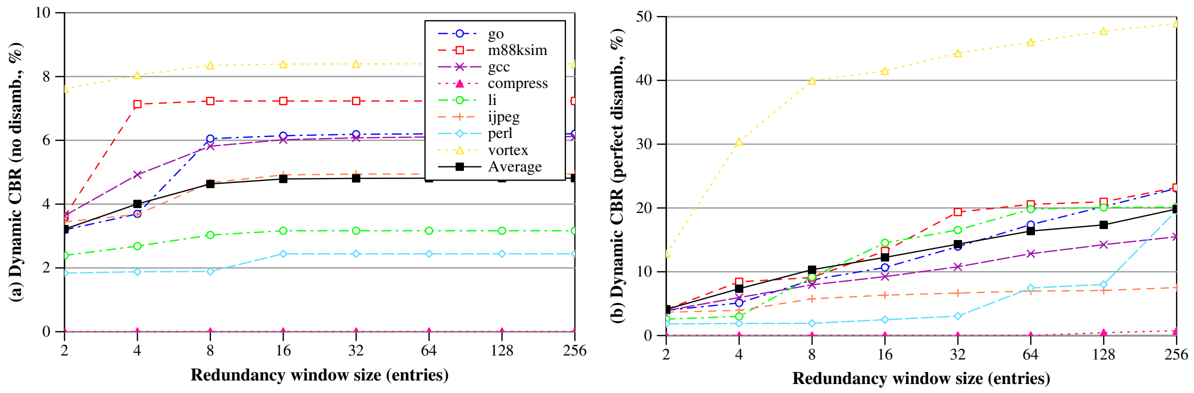


Figure 6.2: Dynamic amount of conditional branch redundancy, assuming: (a) no memory disambiguation, and (b) perfect memory disambiguation (X-axis is logarithmic).

The results of our experiments are shown in Figure 6.2, where we present data for our original benchmark set and for various redundancy window sizes[2]. We present these results in two different graphs:

- Figure 6.2a corresponds to the redundancy observed assuming *no dynamic memory disambiguation* mechanism at all. This means that every value loaded from memory has no previous information associated to it, even if this information would happened to be available in the redundancy window. This situation might occur, for example, when the same value was loaded from memory and evaluated by a conditional branch in the near past. Under this assumption, only around 5% of conditional branches are found to be redundant, which points out that any effort in eliminating redundant conditional branches will expose very low benefits.

- On the other side, Figure 6.2b shows the redundancy observed when assuming a *perfect dynamic memory disambiguation* mechanism. In this case, values loaded from memory do carry range information if this information still remains in the redundancy window[3]. Clearly, except for program `compress`, a significant amount of redundancy exists even in these highly optimized binaries. As an example, the graph shows that, for program `go`, 23% of all conditional branches can be considered redundant because their outcomes can be derived by looking at a small dynamic instruction stream of only 256 entries. That is, 23% of all dynamic conditional branches have *dynamic correlation* and, therefore, should be candidates to be optimized away by the compiler. Program `vortex` is also another interesting example. In this program, almost 50% of all dynamic conditional branches

[2]As we mentioned in Section 3.3, programs were compiled with full optimizations using the Compaq/Alpha C compiler. Similar levels of redundancy have been observed using other compilers, like GNU `gcc`.

[3]There is an exception to this rule. The value range information associated to memory locations is reset on every system call that is known to modify the user memory.

are redundant. We found that more than 30% of redundancy comes from loading and evaluating a variable called `Status` in functions `ChkGetChunk` and `MemGetWord`, by using a pointer which is an input argument of these functions. Since variable `Status` almost never changes, all related branches are considered dynamically redundant.

In general, around 20% of all conditional branches are found to be redundant by looking at the instruction stream within the most recent 256 conditional branches. Today's optimizing compilers [Muc97a] are able to deal with regions larger than this size and, thus, should be expected to optimize all this redundancy away.

It's also interesting to note that, unlike the case of load redundancy, but similarly to the store redundancy results (see Section 5.2.1 and Section 5.2.2, respectively), the obtained level of conditional branch redundancy mostly depends on the program that is being measured.

6.3 CBRE on executable code

As we have already defined in Section 6.1, a conditional branch has *static correlation* along a path if its outcome can be determined along the path at compile time from prior statements or branch outcomes. Such conditional branch is then said to be *redundant* along the correlated path, and can be removed in order to speed up a program. We call this optimization *Conditional Branch Redundancy Elimination (CBRE)*.

Recent research in branch prediction [Kra94, Pat95, YGS95, SLM96] and elimination of conditional branches [MW95, BGS97], as well as the results presented in the previous section, have reported the existence of significant amounts of conditional branch correlation, presenting opportunities for CBRE. We next outline the algorithms used for applying CBRE in the most simple cases, when no control-flow restructuring is needed for removing conditional branches.

6.3.1 Eliminating close redundancy

The results presented in Figure 6.2b show that around 10% of the conditional branch redundancy detected can be captured by using a redundancy window of just 8 entries. This indicates that the first source of redundancy that we should focus our optimization on is located within small groups of basic blocks.

The simplest example of CBRE is shown in Figure 6.3, where we look for short-distance redundancy within an *Extended Basic Block (EBB)*. From the figure, a conditional branch B_1 is taken if the contents of register r_1 is zero. Furthermore, this branch is followed after some instructions by another conditional branch B_2 within the same EBB, whose branch condition is *subsumed* by previous branch B_1. In the example, this means that contents of register r_1 is known to be greater or equal than zero, since it was determined to be zero in B_1. If register r_1 is not modified between both instructions, we can say that B_2 is redundant. Once

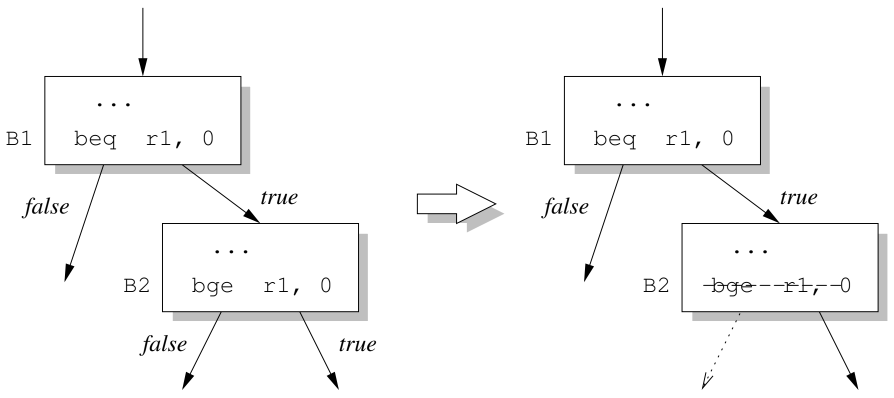


Figure 6.3: Elimination of a redundant conditional branch within an extended basic block.

a redundant conditional branch has been identified, we can eliminate it by simply removing both the redundant branch and the edge for the infeasible path. The expectation is that, after running CBRE, some basic blocks can be merged, and also some others can be removed since they become unreachable.

Although this is the most simple case of CBRE, it already introduces the most important issues that this optimization has to deal with:

1. In the above example we have identified a source of correlation from branch B_1, but correlation may also come from constant assignments, register copies, common subexpressions, etc. Our analysis for discovering branch correlation uses the *Value Range Propagation (VRP)* algorithm we presented in Section 6.2.1.

2. In Section 6.2 we pointed out how important memory disambiguation is in order to catch branch redundancy. We could have formulated our CBRE algorithms to explicitly deal with disambiguation of memory references, but it would have increased the complexity of this formulation unnecessarily. On the contrary, we partially address this issue in our implementation by applying before CBRE the MRE optimizations we presented in Chapter 5. As a result, some of the information propagated by CBRE is not lost anymore when loading data on redundant loads, since most of these unnecessary reloads have been eliminated by MRE.

6.3.2 Eliminating distant redundancy

The CBRE approach described in the previous section was targeted at exploiting close redundancy. However, looking back to Figure 6.2, there is still a significant amount of redundancy that can be caught if we could explore larger distances between instructions. Of course, in order to catch this distant redundancy, we need to apply CBRE to regions of code that expand beyond an EBB and which, therefore, may contain complicated control-flow structures.

The major difference with the previous section is that when working on a candidate branch to be removed, we need to examine *all* possible control-flow paths that may reach the candidate, in order to decide whether it is truly redundant or not. The algorithm for long-distance CBRE proceeds then as follows:

1. First, an initialization phase computes the *slice* of every branch in a function. Corresponding range information is then initialized for every value within the branch slices. These values include both output of instructions and summary information at join of paths in the function.

2. Next, an iterative data-flow analysis propagates value ranges back and forth, similarly to the technique used for short-distance CBRE, until a fixed point is reached. Although our algorithm is flow sensitive, the analysis treats paths together whenever paths meet, by summarizing value ranges applying the join operator (in our case, the union of ranges).

3. Finally, every conditional branch in the function is analyzed for correlation, by simply consulting whether the final range of the branch source register is subsumed by the condition. In such a case, both the branch and the edge for the subsumed path can be eliminated.

Unlike other traditional data-flow algorithms, such as constant propagation, loops need to be handled explicitly when propagating value ranges. To this end, we use the approach proposed by Patterson [Pat95], which identifies ϕ-functions where one or more of the use-def chains are back edges (as identified by a depth first traversal of the flow graph). Value ranges for these *loop carried expressions* will be derived by following simple heuristics, and then marked so that their ranges are not re-evaluated at all.

The above CBRE scheme is applied for every function in the program. The algorithm is quite efficient, since it performs intraprocedural CBRE for all the branches within a function at a time. Besides, because our analysis is restricted to discovering useful information on relevant ranges, we are able to achieve polynomial analysis time in practice [ABD+02, DLS02].

6.4 Path-sensitive profile-guided CBRE

Information about the program execution behavior can be very useful in optimizing programs (see Section 2.3.2). Our proposal is to be aware of *profile information* to guide CBRE. Profile data consists of a frequency for each basic block and a probability for each branch in the program. We next outline how the algorithms presented in Section 6.3 have been adapted for using information gathered in a profile run to (a) choose the candidates for removal, and (b) remove conditional branch redundancies in a path-sensitive fashion.

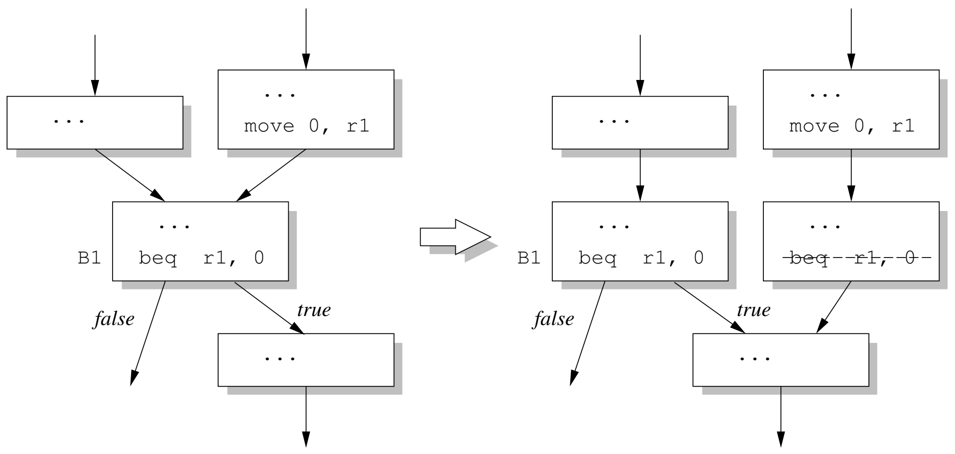


Figure 6.4: Elimination of a redundant conditional branch in a path-sensitive fashion across extended basic blocks. Control-flow restructuring is needed, by applying code replication.

6.4.1 Eliminating close redundancy

CBRE can be seen as a particular case of partial redundancy elimination (PRE). However, the code motion techniques useful for PRE of assignments [KRS94a, CCK+97] do not suffice for removing conditional branches in the general case. We can see an example in Figure 6.4. Branch instruction B_1 has static correlation from its right predecessor path (i.e., the one that contains the "move" instruction), but not from the left one. Due to this "partial redundancy", the algorithms presented in Section 6.3 are not able to remove this particular type of branch redundancy. To eliminate the conditional, unlike techniques used for general PRE, *control-flow graph restructuring* is required in order to separate the correlated path from the rest of the paths. Thus, the path is isolated by applying *code replication*, so that the conditional on this path can become fully redundant and be then removed. The example in Figure 6.4 also shows how this transformation works.

We have adapted the algorithm targeted to catch short-distance redundancy presented in Section 6.3.1 to deal with profile information and path-sensitive redundancy across extended basic blocks. First, to keep the final code growth due to code restructuring under control, we apply CBRE only to "hot branches" in the program. Then, and most importantly, every incoming edge reaching the EBB root of the considered branch is analyzed separately (actually, we only analyze the "hot edges", for the same reason). As each one of these incoming edges identifies a new EBB, our path-sensitive short-distance CBRE performs the same analysis presented in Section 6.3.1, but now "merging" both EBBs as if they were a single one (we use the same heuristic presented in Section 6.3.2 for handling loops). When static correlation is detected on some of these paths, we replicate the code from the original EBB root down to the considered branch, just as the example in Figure 6.4 shows.

This optimization should be performed after a single pass of short-distance CBRE, to remove as many fully redundant branches as possible without the need to replicate code. Also, as code restructuring will insert a significant number of unconditional branches, it would be interesting to run path-sensitive CBRE coupled to an optimization for removing unconditional branches [MW92].

6.4.2 Eliminating distant redundancy

To remove path-sensitive redundancy, unfortunately, the optimizer has to pay the *exponential* price of optimizing each path separately. The reason why analyzers avoid this situation is that there is an exponential number of paths, even in a program with no loops [Bod99]. To stay practical, analyzers (such as the one we presented in Section 6.3.2) treat paths together by summarizing their results whenever paths meet, therefore diluting optimization opportunities that could be exploited otherwise.

As we have seen in the above section, to eliminate branch redundancies we have to separate the correlated paths from the rest of the paths via code duplication. The key to our proposal is to extend the analysis presented in Section 6.3.2 so that isolated paths can be handled separately during analysis time. With this ability, the loss of information that occurs when "merging" value ranges at control-flow joins is avoided, thus increasing the accuracy of the analysis. Then, when a redundancy is detected, code can be effectively duplicated in order to remove such redundancy.

Unlike the approach presented in Section 6.3.2, the path-sensitive CBRE presented here will analyze every considered branch in isolation. The algorithm proceeds as follows:

1. First, we apply the analysis presented in Section 6.3.2 to the initial slice of the candidate branch. The initial slice will only consider the existing path between the EBB root down to the conditional branch. When correlation is detected for a given path, the algorithm marks such path for later duplication. Otherwise, correlation analysis is applied recursively for the resulting slices after adding every incoming edge separately (similarly as we did in the above section), until either (a) the entry point of the function is reached, or (b) value ranges do not depend on ϕ-functions beyond the current slice.

2. Then, for all the marked paths within a function, we "merge" paths in order to only duplicate the necessary code when later removing branch redundancies. *Path merging* will create new paths resulting from "matching" and "fusing" common subpaths, then eliminating the original subsumed paths for duplication.

3. Finally, we perform code duplication on the resulting paths. Corresponding conditional branches on these cloned paths can be removed, since they are now fully redundant.

CBRE	Description	Section
MRE	Complete-MRE	Chapter 5
Local	CBRE within EBBs	6.3.1
Global	Long distance CBRE	6.3.2
Complete	Path-sensitive local-CBRE Path-sensitive global-CBRE	6.4.1 6.4.2

Table 6.1: Description of the different CBRE algorithms under evaluation.

As we did for short-distance CBRE, this optimization will be applied after a phase of long-distance CBRE, so that fully redundant branches can be previously removed.

Due to the exponential price of dealing with each path separately, the above scheme for path-sensitive CBRE does not come without a cost. Since our current implementation uses a backtracking algorithm, we use a couple of simple but effective heuristics in order to reduce the high computational cost of the algorithm:

1. We only apply CBRE to the most important branches (that is, those branches located in the most frequently executed paths of the program).

2. For every considered incoming edge when looking for path-sensitive branch correlation (a) we consider only *hot* edges (i.e., those ones located in the *hot path* of the function), and (b) we do not allow the analysis to follow more than K incoming edges, where K is a fixed constant. In our implementation we have used a K value of 10.

We have observed that by using these heuristics the algorithm misses only a few opportunities for detecting redundancies, while the compilation time does not significantly increase.

6.5 Evaluation

In this section, we describe the process we have followed for evaluating the effectiveness of the CBRE techniques proposed in this chapter.

The CBRE techniques presented in previous sections are general enough to be implemented into a production compiler. However, to take advantage of the benefits of optimizing at the binary level (see Section 1.1.2), we have implemented the proposed CBRE approaches presented in Table 6.1 on the Alto framework we described in Section 3.2.1. The algorithms have been integrated with the rest of optimizations carried out by Alto, such as constant/copy propagation, dead/unreachable code elimination, inlining, etc. This approach ensures not only exposing benefits coming from CBRE itself, but also enhancing the effect of the rest of Alto optimizations.

The integration of the CBRE algorithms has been performed within the optimization scheme presented in Section 3.2.1.1, by following the next guidelines:

- First, we apply complete-MRE (see Chapter 5 and Table 6.1) as a basic form of CBRE, since we are interested in measuring the effects of MRE itself in removing conditional branch redundancy. Besides, MRE will have the effect of maximizing the number of CBRE opportunities by removing memory dependencies, which is a similar effect than letting CBRE explicitly deal with memory disambiguation (see Section 6.3.1).

- We include into the base optimizations the short-distance CBRE within extended basic blocks (i.e., local-CBRE, see Table 6.1). The reason is that, since computing the value ranges and performing the redundancy searches for an EBB is relatively cost-effective, it can be applied several times during the optimization process.

- Within the one-time optimizations, we perform only once the long-distance CBRE algorithm (i.e., global-CBRE, see Table 6.1). This is not an expensive optimization by itself, but it comes after performing some space- and time-intensive data-flow analyses.

  Since the formulation of our algorithms is intraprocedural, procedure inlining is previously performed by default so that our CBRE proposals can have an interprocedural behavior. Besides, unlike other techniques [BGS97], this approach ensures that branch redundancy which is discovered by simply applying constant propagation after inlining is already removed away from the baseline, before evaluating our CBRE methods.

- The local- and global-CBRE optimizations can be also applied in a path-sensitive fashion (i.e., complete-CBRE, see Table 6.1). To keep the running time of the corresponding CBRE algorithm under control we use a ϕ value of 0.75 (see Section 3.2.1.3). The idea is to apply CBRE only to the "hot branches" in the program, but also to decide which paths are hot (see Section 6.4), and be aware of excessive code growth due to code duplication.

  Since complete-CBRE tends to insert a lot of unconditional branches that might increase branch misprediction, we also include in the base transformations a profile-guided optimization for replicating basic block tails in order to remove unconditional branches, which is a simplified version of the one proposed by Mueller and Whalley [MW92].

The benchmarks we have used for our experiments were presented in Section 3.3, and they were generated following the methodology described in Section 3.4. For our experiments, we have chosen the Alto/Inline binaries as a baseline (that is, the most optimized binaries we have), whose characteristics and generation procedure were described in Section 3.3.1.

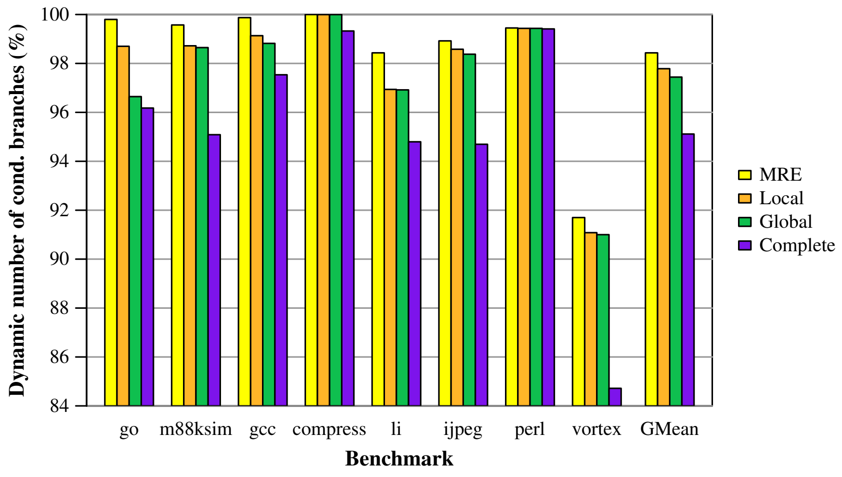


Figure 6.5: Effect of different CBRE algorithms in number of conditional branches at run time. The baseline is Alto/Inline binaries without any CBRE at all.

6.5.1 Reduction in number of dynamic conditional branches

We start evaluating the effectiveness of the CBRE algorithms under study by comparing the number of dynamic conditional branches executed with respect to the program baseline. To get these results, we ran our benchmark suite using the corresponding simulation inputs (see Table 3.2 in Section 3.3) on top the `sim-profile` simulator of the SimpleScalar toolset [ALE02].

Figure 6.5 presents the reduction in number of dynamic conditional branches for every benchmark. As it can be seen, all programs do show improvements typically around 1–5%, with program `vortex` being the best case (up to a 15% reduction). Programs `compress` and `perl` are the worst cases, mainly due to the fact that they were the programs with lower levels of short-distance redundancy (as we saw in Figure 6.2). We can also observe that MRE in isolation is able to achieve a 2% reduction in number of conditional branches, even without considering any explicit conditional branch removal algorithm. This result points out how important memory disambiguation is in order to catch branch redundancy. Also, both local- and global-CBRE only yield an additional 1% reduction, proving that control-flow graph restructuring becomes necessary to effectively increase the number of conditional branches removed. This assumption is confirmed by looking the results for complete-CBRE, that achieves more than a 2% additional reduction.

Our results are consistent with those observed by Bodík, Gupta and Soffa [BGS97], although we can see that we achieve less benefits. We believe the reason is that they catch a lot of interprocedural redundancy that could be also discovered by simply applying constant propagation after function inlining. Since this is what we do in our baseline, we found that this type of redundancy has already been optimized away before applying CBRE.

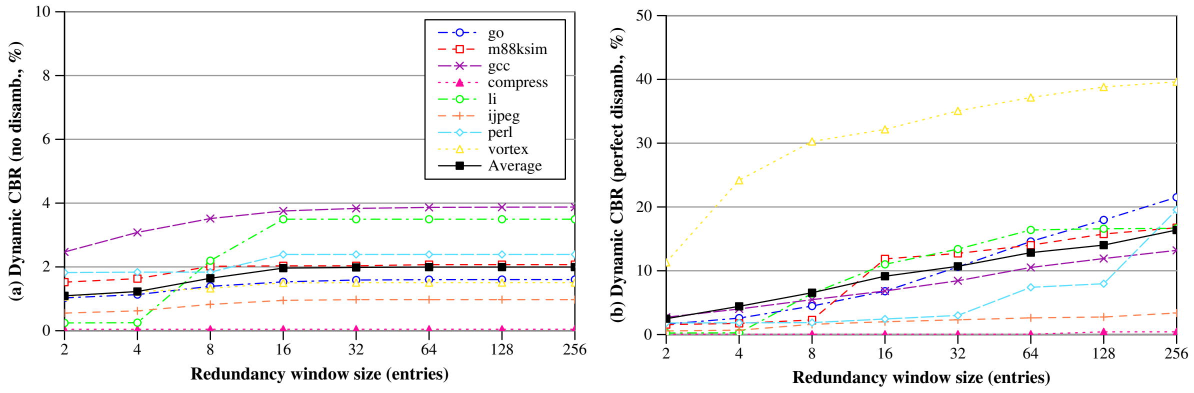


Figure 6.6: Dynamic amount of conditional branch redundancy after complete-CBRE applied, assuming: (a) no memory disambiguation, and (b) perfect memory disambiguation. (X-axis is logarithmic).

Finally, Figure 6.6 presents two graphs showing the percentage of conditional branch redundancy remaining after applying complete-CBRE, which can be compared to the results presented in Figure 6.2 (see Section 6.2). As we can see, CBRE achieves around 3% and 5% reduction in conditional branch redundancy, assuming (a) no memory disambiguation, and (b) perfect memory disambiguation, respectively.

6.5.2 Effects of procedure inlining on CBRE

Since the formulation of our CBRE algorithms is intraprocedural, procedure inlining is performed by default so that our CBRE proposals can have an interprocedural behavior and get benefit from that. However, it would be interesting to measure how avoiding procedure inlining will affect CBRE in "hiding" conditional branch redundancy to be discovered and eliminated. This does not only include interprocedural redundancy, but also intraprocedural redundancy discovered after obtaining interprocedural value range information.

Table 6.2 presents the different benchmark sets we use for evaluating the effect of inlining on CBRE, which is done by measuring the reduction in number of dynamic conditional branches presented in Figure 6.7. These binaries result from enabling/disabling CBRE and procedure inlining in Alto when optimizing our benchmark suite. The *Base* and *Inline* categories correspond to the Alto/Base and Alto/Inline sets we already evaluated in Section 3.3.1. Actually, data in Figure 6.7 for these sets was already presented in Figure 3.4[4]. Category *noInline* means that complete-CBRE is performed with no previous procedure inlining at all. However, as we will explain next, inlining by itself is able to remove some conditional branches, which makes it difficult to make comparisons between programs with/without inlining. Therefore, we consider the *noInline'* category be the same as *noInline*, but we subtract

[4]To be consistent with previous sections, we maintain the Alto/Inline binaries as the baseline in Figure 6.7.

Category	Description	Sections
Base	no CBRE, no inlining (i.e., Alto/Base)	3.3.1
Inline	no CBRE, inlining (i.e., Alto/Inline)	3.3.1
noInline	Complete-CBRE, no inlining	6.5.2
noInline'	Complete-CBRE, no inlining	6.5.2
Complete	Complete-CBRE, inlining	6.5

Table 6.2: Description of the binaries obtained with/without applying CBRE and inlining. The baseline will be *Inline* binaries (i.e., Alto/Inline binaries without any CBRE at all.)

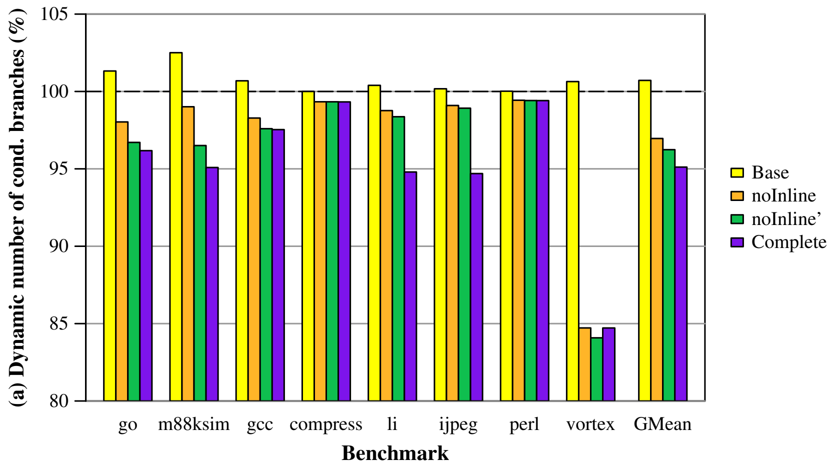


Figure 6.7: Effect of procedure inlining on CBRE, measured as number of conditional branches at run time. The baseline is Alto/Inline binaries without any CBRE at all.

from the final number of conditional branches those that where removed thanks to inlining in the baseline programs, without any CBRE at all. We perform such "trick" since it would be very unfair to compare removed branches across different baselines. Finally, *Complete* category corresponds to the complete-CBRE already presented in Figure 6.5 (inlining included), and it has been added to the figure just for comparison purposes.

From the results presented in Figure 6.7 we can observe first that programs on the *Base* benchmark set execute more conditional branches than the baseline (i.e., relative numbers are higher than 100%). The reason is that, when inlining is applied, some conditional branches can now be removed by simply propagating constants. As a result, Alto/Inline binaries already include some form of conditional branch elimination. The fact that this removal (i.e., elimination of unnecessary conditional branches thanks to the application of constant propagation after applying inlining) is already included as a baseline in all our experiments makes our results even more valuable (i.e., the ones presented in Figure 6.5). A second thing to note is that *noInline* binaries achieve better conditional branch reductions than the inlined

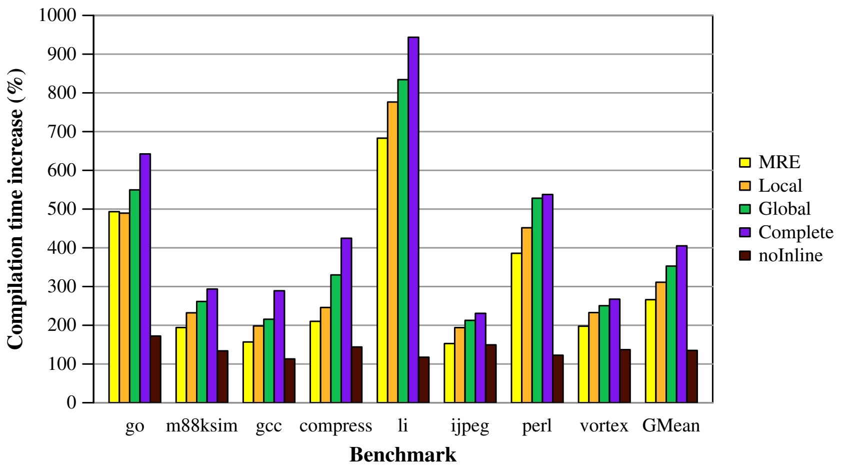


Figure 6.8: Effect of applying CBRE in Alto compilation time, for the SPECint95 programs. The baseline is Alto/Inline binaries without any CBRE at all.

baseline programs. Not only that, but these results are comparable for some programs to those obtained when applying complete-CBRE, and this distance is even reduced when considering the "estimated" *noInline'* programs. This result seems to indicate that, unlike redundant memory references in Section 5.7.3, either these programs do not contain significant amounts of redundant conditional branches across functions, or our algorithms are not powerful enough to discover this interprocedural redundancy.

6.5.3 Compilation time

Related to how procedure inlining affects CBRE, we can see in Figure 6.8 the relative compilation time of Alto when applying the CBRE algorithms as presented in Table 6.1, with respect to our baseline. We also include in the figure the data corresponding to the application of complete-CBRE without any procedure inlining at all. First, Figure 6.8 shows that global CBRE schemes come with an important cost on those programs where inlining was widely applied and the percentage of *hot* basic blocks is significant. In general, procedure inlining dramatically increases CBRE compilation time, while for the non-inlined binaries (i.e., the right bars) the compile-time overhead of CBRE is very moderate. Another interesting observation is that, even although the path-sensitive approach included in complete-CBRE seemed to be very costly (see Section 6.4), the heuristics used prove to be very effective in not increasing compilation time too much with respect the global-CBRE algorithm.

6.5.4 CBRE impact in code growth

Except for complete-CBRE, the rest of CBRE algorithms perform branch removal by actually eliminating both branch instructions and corresponding infeasible paths. This effect comes

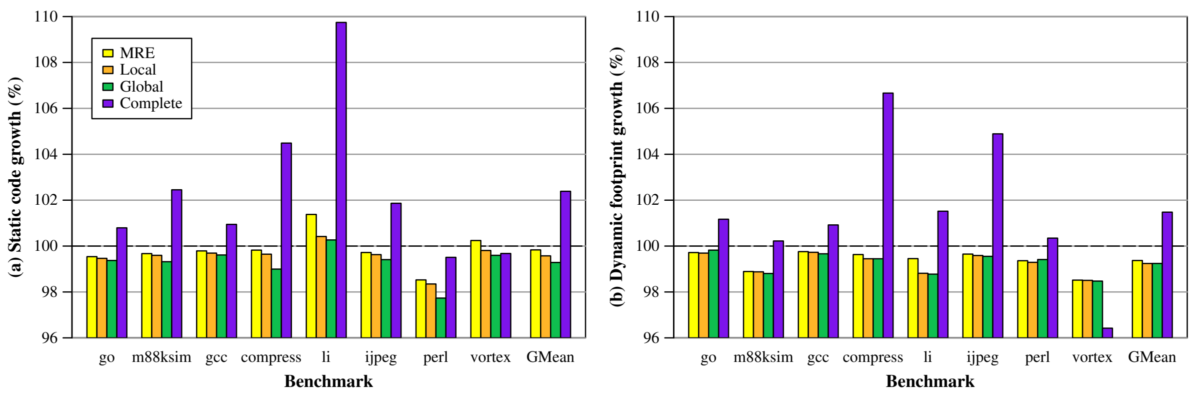


Figure 6.9: Effect of different CBRE algorithms in (a) static code growth, and (b) dynamic footprint growth. The baseline is optimized binaries without any CBRE at all.

with a reduction in code size, which is often beneficial for improving instruction-cache performance. Complete-CBRE, however, replicates code to eliminate conditionals by creating separate paths. Since our approach is based on profile information, we do not estimate the amount of duplicated code before applying complete-CBRE to a particular branch. This is because we *always assume* that the benefits of removing a conditional branch will outperform the possible penalty of code duplication. In order to verify that this is a right assumption, this section presents experimental data about the effects of CBRE in code growth,

Figure 6.9a presents the static code growth (i.e., program size) for each benchmark with respect to the original baseline. First, as we expected, every CBRE algorithm except complete-CBRE reduces code size: around 1% in average, with program `perl` being the best case (up to 2%). As far as complete-CBRE is concerned, we can observe that code size is only increased around a 2%, which is a very moderate growing percentage. The worst cases are programs `compress` (up to 4%), and specially `li` (up to 10%), which is the program where complete-CBRE found a higher percentage of static opportunities for conditional branch removal requiring code duplication.

Code growth is a good measure to evaluate the behavior of complete-CBRE. However, this measure in isolation is not significant for anticipating capacity problems at the instruction cache. A more accurate measure is shown in Figure 6.9b, where we present the dynamic footprint of resulting images after applying CBRE. What this figure shows is the growing percentage of *actual executed code*, which might actually increase instruction-cache misses since it corresponds to real executed instructions at run time. Thus, from Figure 6.9b we can see that programs `compress` and `ijpeg` are the ones increasing their instruction-cache capacity requirements. Whether these requirements incur into performance degradation will also depend on other instruction-cache factors (e.g., locality, final footprint size, etc.). Overall, however, final footprint growth is lower than 2%.

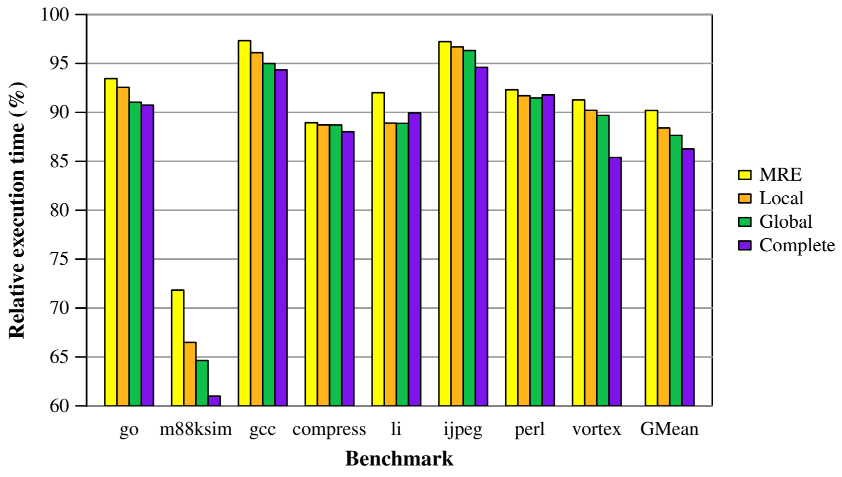


Figure 6.10: Effect of different CBRE algorithms in actual execution time. The baseline is optimized binaries without any CBRE at all.

6.5.5 Speed up using CBRE

Counting the number of removed conditional branches or measuring the static code growth is certainly of interest to understand the effectiveness of every CBRE algorithm. However, a definitive measure of interest is whether execution time is reduced or not. Figure 6.10 presents the relative execution time of our benchmark suite after the different CBRE algorithms, using the execution inputs (see Table 3.2 in Section 3.3). These results were recorded by running the benchmark suite in our target platform, as we described in Section 3.1.3.

From the results presented in Figure 6.10 we can first observe that the most significant reduction in execution time stands for MRE (up to a geometric mean of almost 10%). As we noted in Section 5.7.5, this reduction is mainly due to benefits coming from eliminating redundant memory operations. As far as CBRE is concerned, the local- and global-CBRE approaches only achieve together an additional 2% reduction, being program `m88ksim` the best case. This program and `vortex` are mostly responsible for the additional 1–2% reduction yield by complete-CBRE. Overall, however, the decrease in execution time we observe shows that (a) we have removed some conditional branches that indeed were on the program's critical path and, therefore, contributed to overall execution time, and (b) other optimizations carried out by our binary optimizer do take advantage of CBRE, thus resulting in more efficient code.

It is interesting to note that after applying complete-CBRE, programs `li` and `perl` result in some performance degradation, while reduction in program `ijpeg` is lower than we could expect by looking at the corresponding reduction in dynamic conditional branches in Figure 6.5. In order to explain these effects and also to get an accurate measure of the differences at the processor-core level of applying the CBRE algorithms, we have simulated our benchmark suite on the `sim-alpha` simulator we introduced in Section 3.2.2 [DBK01], using the corresponding simulation inputs (see Table 3.2 in Section 3.3). This out-of-order simulator faithfully models a Compaq/Alpha 21264 configuration that matches our target environment presented in Section 3.1.1.

Two interesting points can be pointed out by analyzing the low-level internal statistics produced by the simulator:

- First, branch misprediction rate was slightly higher when applying CBRE (difference, nonetheless, is under 1% in all benchmarks). This is due to the fact that we are removing correlated conditional branches that were mostly well predicted by the hardware. Thus, less "hit" accesses are requested to the branch predictor, which limits CBRE for achieving better speedups.

- On the other side, the miss-ratio of the L1 instruction cache was also slightly higher when applying complete-CBRE, due to the resulting code growth when the flow graph is restructured (although less than 2% in all benchmarks). Since cache-conscious procedure inlining has already been applied as a baseline (see Section 3.2.1.2), the resulting code footprint for these programs already "fills" the instruction cache in some way. Therefore, even small code growths may hurt the final program performance. We believe this is the actual reason why some programs showed a small slowdown in Figure 6.10.

Finally, the relative execution times we obtained from these simulations are quite consistent with the results presented in Figure 6.10. Small differences are attributed to simulation inaccuracy, as shown in [BS98].

6.6 Related work

While a number of systems have been described for optimization of executable code (see Section 2.4), to the best of our knowledge, any elimination of redundant conditional branches carried out by these systems is limited to fairly simple removal.

Partial Redundancy Elimination (PRE) [MR79, KRS94a, CCK$^+$97, HH97b, GBF98, BGS98, CX03, SHK04] is a well-known scalar optimization that subsumes various *ad hoc* code motion optimizations (such as common subexpression elimination and loop invariant code motion) by attempting to remove redundancies that occur only on some control-flow paths of a program. CBRE can be seen as a particular case of PRE. However, the code motion techniques useful for PRE of assignments do not suffice for removing conditional branches. To eliminate a conditional, *control-flow graph restructuring* is usually required.

The simplest form of branch elimination is loop unrolling [Muc97j], in which instances of back-edge branches are removed by replicating the body of the loop. More sophisticated techniques examine control and data flow simultaneously to identify correlation among branches.

An algorithm for intraprocedural restructuring was first proposed by Mueller and Whalley [MW95], although their technique was mostly focused on eliminating conditionals within loops. A more general approach based upon interprocedural demand driven analysis as well as profile-guided control-flow restructuring was given by Bodík, Gupta and Soffa [BGS97]. However, a lot of interprocedural redundancy they remove could be also eliminated by simply applying constant propagation after function inlining. Mueller and Whalley [MW92] also investigated avoiding unconditional jumps by code replication.

In the spirit of *abstract interpretation* [CC77, CC79] (see Section 2.2.4), several methods have been proposed for obtaining dynamic program properties by using *symbolic evaluation* and simple algebraic rules [Rau91, TP95, ABD+02, DLS02]. Some of these techniques have been focused on a wide variety of program optimizations based on propagating information about value ranges of variables [Pat95, SBA00, BGSW00, MRS+01, CGS04]. In particular, Patterson [Pat95] developed a technique for improving static branch prediction by propagating information about value ranges of variables through a program. The approach we used for discovering conditional branch correlation is a simplified version of his *value range propagation* algorithm.

6.7 Conclusions

In this chapter we have shown that, even although the compiler may have optimized a program aggressively, a significant number of redundant conditional branches appear in the final executable file. This *conditional branch redundancy* does not only appear due to how programmers write source code, but also due to limitations in the compilation model of traditional compilers. To address these issues, we have presented a set of different algorithms to be applied in the context of binary or link-time optimizers, which are targeted to discover conditional branches that are redundant and can be safely removed in order to speed up a program. We call this optimization *Conditional Branch Redundancy Elimination (CBRE)*.

First, we have quantified how much conditional branch redundancy is present in executable programs, and shown that around 20% of conditional branches can be considered redundant from the compiler's point of view because their outcomes can be determined from a previous small dynamic execution frame. We also showed how important memory disambiguation is in order to catch branch redundancy (up to 2% of dynamic conditional branches can be eliminated even without considering any explicit conditional branch removal algorithm). Then, we presented several CBRE algorithms targeted at optimizing away these redundancies:

- The first algorithm (i.e., local-CBRE) is targeted at catching short-distance redundancy within extended basic blocks, while a more general approach (i.e., global-CBRE) works over regions of arbitrary control-flow complexity. Both techniques only achieved small reductions, proving that control-flow graph restructuring becomes necessary to effectively increase the number of conditional branches that can be removed.

- Finally, complete-CBRE extend the above algorithms in a path-sensitive fashion by using program profiles, for identifying partial correlations and performing the necessary code replication to remove branches. This results in a geometric mean reduction of around 2% in both dynamic conditional branches and execution time.

Overall, the observed reductions in execution time show that other optimizations carried out by the binary optimizer do take advantage of CBRE, thus resulting in more efficient code.

We have provided exhaustive experimental results in order to measure the effectiveness of the CBRE algorithms under study, such as accurate measurements of the impact of applying CBRE in code and dynamic footprint growth. Although the final growth is very moderate in both cases, it is responsible of a slightly increase in L1 instruction-cache miss-ratio. We believe this is the reason why, at the end of the day, some programs showed a small slowdown.

Chapter 7

Conclusions and future directions

In this final chapter we summarize what have been the main contributions of this thesis, and present the final conclusions of the results obtained in the previous chapters. We also present some future lines of work opened up by our proposals.

7.1 Introduction

Optimizations targeted at eliminating redundancy are the backbone of compiler techniques for improving program's behavior [Bod99]. As a unifying paradigm, these optimizations expose unnecessary recomputations at program run time of values that are already known, because they have already been computed in the program or because they can be computed at compile time. These recomputations are specially important for memory and conditional branch instructions on binary programs, as proven by the fact that today's microprocessors contain specialized hardware to partially exploit the existing *binary redundancy* for this type of instructions.

Binary redundancy comes partly from the way that programmers write source code, but also from limitations in the compilation model of traditional compilers, which introduces unnecessary memory and conditional branch instructions. Effectively, our observations when measuring dynamic redundancy in Section 5.2 and Section 6.2 suggest that today's optimizing compilers miss numerous redundancy-based optimization opportunities, even in highly optimized programs. Not only that, but also our experiments show that, for most instructions, the source of their redundancy comes only from some although frequent execution paths. Because not all executions of a redundancy are optimizable, this *partial* or *path-sensitive* redundancies are beyond conventional optimizers. Thus, conservative analyzers fail to expose it, and inflexible transformations fail to remove it.

The above facts lead to the increased attention that *binary optimizations* applied at link time or directly to final program executables have received in recent years. As a result, *binary optimizers* overcome most of the existing limitations of traditional source-code compilers, since they can easily optimize the program as a whole or even apply straightforwardly profile-directed compilation techniques without being forced to re-build every source file. However, working at the executable code level turns up some relevant issues that need to be addressed, since most of the high-level information available in traditional compilers is lost.

7.2 Lessons and observations

We next highlight in this section the most important observations made during the development of this thesis.

Programs contain redundancy As we have seen in Section 1.1.1, redundant operations expose unnecessary recomputations at program run time of values that are already known because they have somehow already been computed in the past. This redundancy appears not only because it is introduced by programmers when writing source code, but also due to limitations in the compilation model of traditional compilers. As a result, these redundancies expose optimization opportunities that can be potentially exploited by an optimizing compiler in order to remove them away.

Among the large variety of redundant instructions, we have discussed the two most important classes existing in binary programs: memory and conditional branch redundancy. We have then measured in Chapter 5 and Chapter 6 that, even considering a redundancy window of just 256 entries, around 75% of load references, 10% of store references, and 20% of conditional branches are potentially redundant, which proves that significant amounts of *binary redundancy* exists even in highly optimized programs.

Binary optimizations yield significant benefits As we have seen in Section 1.1.2 and Section 2.4, applying low-level optimizations at a very late stage or even at post-link time may carry significant improvements in the behavior of the optimized programs. The reason is that *binary optimizers* overcome most of the existing limitations of traditional source-code compilers, exploiting new opportunities for program optimization as we have shown in Section 3.3.1, where our framework yields around 14% reduction in execution time. However, we have also seen that some relevant issues appear when working at the executable code level, since most of the high-level information available in traditional compilers is lost.

Low-level alias analysis is difficult While there is an extensive body of work on pointer alias analysis of various kinds, they are mostly high-level analyses carried out in terms of source language constructs. Unfortunately, such analyses turn out to be of limited utility at the machine code level. In fact, as we have seen in Section 2.2.3, the problem of memory disambiguation is one of the weak points of object code modification, because high-level information available in a traditional compiler is lost. Our results have shown that only a 15% of the disambiguation queries can be successfully resolved at compile time. Proposing sophisticated low-level alias analyses become then necessary, since features such as pointer arithmetic and out-of-bounds array accesses must be handled at this level, where the contents of every register is potentially an address.

Binary redundancy can be eliminated Taking advantage of the binary optimization technology, but also addressing its corresponding issues, we have developed in Chapter 5 and Chapter 6 profile-guided optimizations targeted at eliminating memory and conditional branch redundancies. As a result, a significant amount of binary redundancy can indeed be eliminated, which translates into an important 14% execution time reduction in our benchmark suite.

Path-sensitivity is important To effectively perform sophisticated analyses and optimizations, it is important in most cases to consider path-sensitive information. As our experiments have shown, a significant number of optimization opportunities cannot be exploited by just considering standard data-flow techniques. On the contrary, performing program analyses and transformations in a path-sensitive manner may expose significant benefits, even when using simple profiling schemes (see Section 3.2.1.2).

Perhaps the most important observation we have extracted out of this research is that, even considering the existing hardware structures targeted at exploiting binary redundancy, eliminating this redundancy at a very late compilation stage has always a positive impact in the behavior of the optimized programs.

7.3 Summary of contributions

In this thesis we have presented new *profile-guided compiler optimizations* for eliminating the existing redundancy encountered on executable programs at binary level. Our *Binary Redundancy Elimination (BRE)* techniques are targeted at eliminating both *redundant memory operations* and *redundant conditional branches*, which are the most important ones for addressing the major performance issues in today's microprocessors. Particular emphasis was placed on implementing our proposals within a *binary optimizer*, which overcomes most of the existing limitations of traditional source-code compilers. However, since most of the high-level information is lost when optimizing binaries, we have also pointed out the most relevant issues of applying our algorithms at the executable code level.

Within this scenario, the most important contributions of our work roughly correspond to the different chapters of this document, which we consider to be the following:

Computing alias information We have first reviewed in Chapter 4 the problem of *alias analysis* at the executable program level, identifying why memory disambiguation is one of the weak points of object code modification. Then, we presented several alias analyses to be applied in the context of link-time or executable code optimizers. First, we have proposed a *must-alias* analysis to recognize memory dependencies in a *path-sensitive* fashion, which increases alias accuracy a 50% over a path-insensitive scheme. Then we introduced two *speculative may-alias* data-flow algorithms to recognize memory independencies. These may-alias analyses are based on introducing unsafe speculation at analysis time, which increases alias precision on important portions of code and keeps the analysis reasonably cost-efficient. Our results have shown that our analyses prove to be very useful for increasing memory disambiguation accuracy of binary code, up to 83% in average, which turns out into opportunities for applying optimizations.

Eliminating memory redundancies We have discussed in Chapter 5 the discovery and elimination of memory operations that are redundant and can be safely removed in order to speed up a program, an optimization that we call *Memory Redundancy Elimination (MRE)*. Quantifying these effects we have shown that a high percentage of memory references at program run time can be considered redundant because they are accessing memory locations that have been referenced in a near past. Then, we presented several profile-based MRE algorithms targeted at optimizing away these redundancies, which are based on PRE techniques for eliminating partial redundancies in a path-sensitive fashion. Our results have shown that a significant amount of memory redundancy can indeed be eliminated, up to 18% of load redundancy and 8% of store redundancy, which translates into an important 10% reduction in execution time.

Eliminating conditional branch redundancies Finally, in Chapter 6 we have proposed different optimizations for detecting an eliminating redundant conditional branches on executable code. These are branches whose outcome can be determined at compile time, and thus they can be safely removed in order to speed up a program, We call this optimization *Conditional Branch Redundancy Elimination (CBRE)*. We then presented several CBRE algorithms targeted at optimizing away these redundancies, based on the observation that important amounts of conditional branches in a program can be considered redundant because their outcomes can be determined from a previous short dynamic execution frame. The key ideas of our proposed CBRE algorithms result from combining control-flow restructuring and profile information. We also pointed out how important memory disambiguation is in order to catch branch redundancy. Our results have shown that a around a 5% of conditional branches can be removed with moderate levels of code growth.

From the above contributions and the work presented in this dissertation, we have produced several technical reports [FED01a, FE02a, FE03, FE04a] and publications [FED01b, FE02b, FED03, FE04c, FE04b].

7.4 Future directions

The research described in this dissertation suggest several avenues for future work. The most obvious of these future directions would explore extensions to our proposals.

Extend our proposed alias analyses We mentioned in Section 4.2 that our current implementation of path-sensitive memory disambiguation uses a backtracking algorithm. However, we should provide a data-flow formulation of the analysis so that the problem of path-sensitive disambiguation can be more straightforward established by using an systematic iterative algorithm.

It would be also interesting to implement the speculative alias analyses we developed in Section 4.3, within a production compiler [BCC+00] containing speculative optimizations based on reordering memory operations [HSS94, GCM+94, MM97, BCC+00, PGM00a]. Besides solving the possible problems of adapting our speculative analyses, the idea would be to test whether they are beneficial in performing decisions for speculative optimization opportunities.

Consider additional MRE techniques We have seen in Chapter 5 that our MRE techniques were able to remove a significant percentage of memory redundancy. However, a lot of redundancy still remains in our programs after applying MRE. For instance, redundant sequential array accesses will be neither recognized nor removed by using our schemes. It would be interesting to consider using other techniques for removing memory redundancy, such as scalar replacement [CCK90], to fully obtain the potential of the MRE optimizations.

Improve CBRE optimizations The value range propagation (VRP) algorithm we presented in Section 6.2.1 for detecting branch correlation is a simplified version of a VRP framework proposed by Patterson [Pat95]. We observed that, even although our current VRP is able to discover an important number of static correlation among branches, this number is not significant enough to fully discover the existing branch redundancy. We should consider instead either a better VRP framework in order to propagate more accurate value ranges [Pat95, SBA00, BGSW00, MRS+01], or even a more general frameworks based on abstract interpretation (see Section 2.2.4).

We have also seen in Chapter 6 that our CBRE optimizations do not suffer from excessive code growth due to code duplication. However, the more CBRE opportunities discovered and eliminated, the more critical code growth will be. It would be then convenient to propose *code-growth conscious CBRE* techniques, to keep the final code growth due to code restructuring under control.

We consider the above proposals for future work to be the next step of further research in the area of binary redundancy. Furthermore, we next discuss several proposals leading towards the BRE goals, that might be a step ahead of future work.

Explore additional alias analyses We have seen in this dissertation that having accurate aliasing information is key to effectively detect and eliminate binary redundancy, even although performing good-quality alias analysis is one of the weak points of optimizing binary code. Therefore, we think that proposals presented in Chapter 4 are not enough and we should further consider exploring better alias analysis algorithms to fully obtain the potential of the BRE optimizations. For example, promoting pointer aliasing information from the high-level stages of the compiler down to the executable code optimizer would significantly improve alias accuracy.

Consider new path-sensitive mechanisms In previous chapters we have seen that, given the inevitable exponential cost, the imperative for path-sensitive analysis and optimization is to exploit individual program paths only as far as it is practical. We have solved this problem by applying simple heuristics to reduce the exponential cost of the path-sensitive algorithms. However, we think that it would be interesting to consider more sophisticated path-based optimization models [BA98, Bod99] and profiles [BL96, BMS98], in order to reduce the cost and boost the benefits of removing binary redundancy. Furthermore, it would also be interesting to extend our BRE algorithms beyond procedure boundaries, obtaining *interprocedural BRE optimizations*.

The promising results presented in this thesis show that significant potential exists for eliminating binary redundancy. Future processors should perhaps combine hardware prediction, which identifies dynamic redundancy, with compiler analyses and/or transformations at run time, so that total program execution time can be reduced.

Bibliography

[ABD$^+$97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997. 2.4.2, 3.2.2

[ABD$^+$02] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. *Lecture Notes in Computer Science*, 2477:230–246, 2002. 2.2.4, 6.2.1, 6.3.2, 6.6

[ABZT98] Wolfram Amme, Peter Braun, Eberhard Zehendner, and François Thomasset. Data dependence analysis of assembly code. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–347, Paris, France, October 12–18 1998. 2.2.3.2, 4.4

[ACD74] T. L. Adam, K. M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974. 2.3.1.6

[AGS97] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 134–145, Las Vegas, Nevada, June 16–18 1997. 2.3.1.4, 3.2.1.2

[AJ88] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 241–249, Atlanta, Georgia, June 20–24 1988. 2.3.1.4

[AK00] Hakan Aydin and David R. Kaeli. Using cache line coloring to perform aggressive procedure inlining. In *Proceedings of the 4th Workshop on Interaction be-*

tween Compilers and Computer Architectures, Toulouse, France, January 2000. 2.3.1.4

[AKS00] Erik R. Altman, David R. Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3):40–45, March 2000. 2.4

[AL98] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 17–19 1998. 4.4

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002. 3.1.1, 3.2.2, 5.7.1, 6.5.1

[ASU86a] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools*, Chapter 10: Code Optimization, pages 585–722. In [ASU86c], 1986. 1.1.1, 1, 2.2.1, 2.2.2, 2.2.3, 2.3, 2.3.1.2, 2.3.1.3, 2.3.2, 2.3.3, 2.3.3.1, 4.3, 5.3, 5.6.2

[ASU86b] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools*, Chapter 1: Introduction to Compiling, pages 1–24. In [ASU86c], 1986. 1.1.1, 2.4.1

[ASU86c] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986. 2.1, 7.4

[ASU86d] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools*, Chapter 8: Intermediate Code Generation, pages 463–512. In [ASU86c], 1986. 2.1.3, 3.1.1

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, Computer Sciences Department, University of Wisconsin-Madison, 1997. 4.3.4.2

[BA98] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 237–251, San Diego, California, January 19–21 1998. 2.3.3.1, 4.3, 4.4, 5.6.1, 5.8, 7.4

[BCC$^+$00] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflehner, Kishore Menezes, Kalyan Muthukumar, and Jim Pierce. The Intel IA-64 compiler code generator. *IEEE Micro*, 20(5):44–53, September/October 2000. 4.3.3, 7.4

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, Canada, June 18–21 2000. 2.3.2

[BG97] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 159–170, Las Vegas, Nevada, June 16–18 1997. 2.3.3

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, California, March 23–26 2003. 2.3.2

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. 2.2, 2.3

[BGS97] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditonal branch elimination. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 146–158, Las Vegas, Nevada, June 16–18 1997. 2.3.3, 2.3.3.3, 6.1, 6.3, 6.5, 6.5.1, 6.6

[BGS98] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 1–14, Montreal, Canada, June 17–19 1998. 1.1.1, 2.3.3, 2.3.3.1, 5.8, 6.6

[BGS99] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 64–76, Atlanta, Georgia, May 1–4 1999. 2.3.3.2, 5.8

[BGSW00] Mihai Budiu, Seth Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 6th International EuroPAR Conference*. Springer-Verlag, August 2000. 2.2.4, 6.2.1, 6.6, 7.4

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 2–4 1996. 1.1.2, 2.3.2.1, 7.4

[BMS98] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 46–57, San Diego, California, January 19–21 1998. 1.1.2, 2.3.2.1, 3.2.1.2, 7.4

[Bod99] Rastislav Bodík. *Path-Sensitive, Value-Flow Optimizations of Programs.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1999. 1.1.1, 2.3.3.1, 4.2, 5.6.1, 5.6.1, 5.8, 6.1, 6.4.2, 7.1, 7.4

[BS98] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *Computer*, 31(5):41–49, May 1998. 3.2.2, 5.7.6, 6.5.5

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 17–19 1977. 2.2.3.2, 2.2.4, 6.6

[CC79] P. Cousot and R. Cousot. Semantic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, January 29–31 1979. 2.2.4, 6.6

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990. 2.3.3.2, 5.8, 7.4

[CCK$^+$97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, Nevada, June 16–18 1997. 2.3.3.1, 5.8, 6.4.1, 6.6

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 152–161, Palo Alto, California, July 25–27 1986. 2.3.1.1

[CE00] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, March 2000. 2.4

[CFE97] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–269, Research Triangle Park, North Carolina, December 1–3 1997. 1.1.2, 2.3.2.1, 4.4

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):452–490, October 1991. 2.2.2

[CG94] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, Chicago, Illinois, April 18–21 1994. 1.1, 6.1

[CGL97] Robert Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing Alpha executables on Windows/NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997. 2.4.2

[CGLR97] Robert Cohn, David W. Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows/NT Workshop*, pages 17–23, Seattle, Washington, August 11–13 1997. 1.1.2, 2.4.2

[CGS04] Ramon Canal, Antonio González, and James E. Smith. Software-controlled operand-gating. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 125–136, San Jose, California, March 20–24 2004. 2.2.4, 6.2.1, 6.6

[CH95] Paul Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 46–56, La Jolla, California, June 18–21 1995. 2.3.1.1

[CH00] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 57–69, Vancouver, Canada, June 18–21 2000. 2.2.3, 4.1, 4.4

[Chi01] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN 2001*

Conference on Programming Language Design and Implementation, pages 191–202, Snowbird, Utah, June 20–22 2001. 2.3.2.1

[CHK93] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, February 1993. 2.3.1.4

[CKJA98] Brad Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, California, October 2–7 1998. 2.3.2.1

[CL96] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 80–89, Paris, France, December 2–4 1996. 1.1.2, 2.4.2, 4.4

[CL97] Keith Cooper and John Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, Nevada, June 16–18 1997. 2.3.3.2, 5.8

[CLCG00] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, December 2000. 2.3.2

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992. 1.1.2, 2.3.1.4, 3.2.1.2, 4.4

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, S. Margherita Ligure, Italy, June 22–24 1995. 1

[Com99a] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*. Number EC-RJ66A-TE. 1999. 3.1.1

[Com99b] Compaq Computer Corporation. *Compiler writer's guide for the Alpha 21264*. Number EC-RJ66B-TE. 1999. 3.3.1.3, 8

[Con97] Daniel A. Connors. Memory profiling for directing data speculative optimizations and scheduling. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, May 1997. 2.3.2.1, 4.4

[CR82] A. L. Chow and A. Rudnick. The design of a data-flow analyzer. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 106–119, Boston, Massachusetts, June 23–25 1982. 2.2

[CX03] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, pages 91–102, San Francisco, California, March 23–26 2003. 2.3.3.1, 5.8, 6.6

[dBdSvP04] Bruno de Bus, Bjorn de Sutter, and Ludo van Put. Link-time optimization of ARM binaries. In *Proceedings of the 2004 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 211–220, Washington, D. C., June 11–13 2004. 2.4.2

[DBK01] Rajagopalan Desikan, Doug Burger, and Stephen Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, Göteborg, Sweden, June 30–July 4 2001. 3.2.2, 5.7.6, 6.5.5

[dBKC$^+$03] Bruno de Bus, Daniel Kaestner, Dominique Chanet, Ludo van Put, and Bjorn de Sutter. Post-pass compactation techniques. *Communications of the ACM*, 46(8):41–46, August 2003. 2.4.2

[DE02] Saumya K. Debray and William Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 95–105, Berlin, Germany, June 17–19 2002. 2.4.1

[Deb95] Saumya K. Debray. Abstract interpretation and low level code optimization. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 111–121, La Jolla, California, June 21–23 1995. 2.2.4

[DEMdS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000. 2.4.1, 2.4.2

[DGR99] Dean Deaver, Rick Gorton, and Norm Rubin. Wiggins/Redstone: An on-line program specializer. In *Hot Chips 11*, Stanford, California, August 15–17 1999. 2.3.2

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997. 4.3, 4.4

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 17–19 2002. 2.2.4, 6.2.1, 6.3.2, 6.6

[DMW98] Saumya K. Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 12–24, San Diego, California, January 19–21 1998. 2.2.3.2, 2.4.2, 4.3.1, 4.4

[DR95] Kayvalia Dixit and Jeff Reilly. SPEC95 questions and answers. *SPEC Newsletter*, 7(3), September 1995. 3.3

[dSdBdB02] Bjorn de Sutter, Bruno de Bus, and Koen de Bosschere. Sifting out the mud: Low level C++ code reuse. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 275–291, Seattle, Washington, November 4–8 2002. 2.4.2

[dSdBdBD01] Bjorn de Sutter, Bruno de Bus, Koen de Bosschere, and Saumya K. Debray. Combining global code and data compaction. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 29–38, Snowbird, Utah, June 2001. 2.3.1.1, 2.4.2

[dSVdBdB03] Bjorn de Sutter, Hans Vandierendonck, Bruno de Bus, and Koen de Bosschere. On the side-effects of code abstraction. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 244–253, San Diego, California, June 11–13 2003. 2.4.2

[DWM98] Amer Diwan, Kathryn S. WcKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 106–117, Montreal, Canada, June 17–19 1998. 2.2.3, 4.1, 4.4

[EAB$^+$02] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. ASim: A performance model framework. *Computer*, 35(2):68–76, February 2002. 3.2.2

[EEF$^+$97] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code compression. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, Nevada, June 16–18 1997. 2.4.1

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24 1994. 2.2

[EGK$^+$94] Kemal Ebcioglu, Randy Groves, Ki-Chang Kim, Gabriel Silberman, and Isaac Ziv. VLIW compilation techniques in a superscalar environment. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 36–48, Orlando, Florida, June 20–24 1994. 2.3.1.6

[FE02a] Manel Fernández and Roger Espasa. Speculative alias analysis for executable code. Technical Report UPC-DAC-2002-27, Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain, 2002. 7.3

[FE02b] Manel Fernández and Roger Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 222–231, Charlottesville, Virginia, September 22–25 2002. 7.3

[FE03] Manel Fernández and Roger Espasa. A combined algorithm for memory redundancy elimination on executable code. Technical Report UPC-DAC-2001-38, Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain, 2003. 7.3

[FE04a] Manel Fernández and Roger Espasa. Link-time optimization techniques for eliminating conditional branch redundancies. Technical Report UPC-DAC-2004-1, Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain, 2004. 7.3

[FE04b] Manel Fernández and Roger Espasa. Link-time optimization techniques for eliminating conditional branch redundancies. In *Proceedings of the 8th Work-*

shop on Interaction between Compilers and Computer Architectures, Madrid, Spain, February 2004. 7.3

[FE04c] Manel Fernández and Roger Espasa. Link-time path-sensitive memory redundancy elimination. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 300–310, Madrid, Spain, February 14–18 2004. 7.3

[FED01a] Manel Fernández, Roger Espasa, and Saumya K. Debray. Load redundancy elimination on executable code. Technical Report UPC-DAC-2001-3, Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain, 2001. 7.3

[FED01b] Manel Fernández, Roger Espasa, and Saumya K. Debray. Load redundancy elimination on executable code. In *Proceedings of the 7th International EuroPAR Conference*, pages 221–229. Springer-Verlag, August 2001. 7.3

[FED03] Manel Fernández, Roger Espasa, and Saumya K. Debray. Load redundancy elimination on executable code. *Concurrency and Computation: Practice and Experience*, 15(10):979–997, August 2003. 7.3

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin-Cummings, Redwood City, CA, 1995. 2.4.1

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compactation. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981. 2.3.1.6, 3.2.1.2

[FLM$^+$01] Richard Flower, Chi-Keung Luk, Robert Muth, Harish Patil, John Shakshober, Robert Cohn, and P. Geoffrey Lowney. Kernel optimizations and prefetch with the Spike executable optimizer. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, Austin, Texas, December 2001. 1.1.2, 2.4.2

[FPP97] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 159–169, Research Triangle Park, North Carolina, December 1–3 1997. 1

[GBF98] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 Interna-*

tional Conference on Computer Languages, pages 230–239, Chicago, Illinois, May 14–16 1998. 1.1.2, 2.3.3, 2.3.3.1, 4.3.3, 4.4, 5.8, 6.6

[GBSC97] Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 303–313, Research Triangle Park, North Carolina, December 1–3 1997. 1, 2.3.1.5

[GCM$^+$94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-Mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, San Jose, California, October 5–7 1994. 4.3.3, 4.4, 7.4

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 47–58, Snowbird, Utah, June 20–22 2001. 1.1.1, 2.2.3, 4.1, 4.4

[GMZ02] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Chapter 4: Profile Guided Compiler Optimizations, pages 143–174. CRC Press, September 2002. 1.1.2, 2.3.2, 2.3.2.1

[Goo97] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 122–133, Las Vegas, Nevada, June 16–18 1997. 1.1.2, 2.2.1, 2.4.2

[HC89a] Wen-Mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, Jerusalem, Israel, May 19–21 1989. 2.3.1.5

[HC89b] Wen-Mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 246–255, Portland, Oregon, June 19–23 1989. 2.3.1.4

[HH97a] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997. 2.4

[HH97b] R. Nigel Horspool and H. C. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *8th Israeli Conference on Computer System and Software Engineering*, pages 111–118, Herzliya, Israel, 1997. 2.3.3, 2.3.3.1, 5.5.1, 3, 5.5.3, 5.6.1, 5.8, 6.6

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Conference*, pages 125–136, January 1992. 2.4

[HMC$^+$93] Wen-Mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993. 2.3.1.6, 2.3.3

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, California, second edition, 1996. 1.1

[HPRA02] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSim: Simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, February 2002. 3.2.2

[HSS94] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 200–210, Chicago, Illinois, April 18–21 1994. 4.3.3, 4.4, 7.4

[Hun00] Robert Hundt. HP Caliper: A framework for performance analysis tools. *IEEE Concurrency*, 8(4):64–71, October 2000. 3.2.2

[JNMW00] Roy Dz-Ching Ju, Kevin Nomura, Uma Mahadevan, and Le-Chun Wu. A unified compiler framework for control and data speculation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 157–168, Philadelphia, Pennsylvania, October 15–19 2000. 1.1.2

[KE93] Danial R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, New Mexico, June 21–25 1993. 2.3.1.6

[Kes99] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, April 1999. 1.1, 3.1.1

[KFML00] AJ KleinOsowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workload characterization of emerging computer applications*, pages 83–100, Austin, Texas, September 16 2000. 3.3

[KK98] John Kalamatianos and David R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 244–253, Las Vegas, Nevada, January 31–February 4 1998. 2.3.1.5

[KMW98] Richard E. Kessler, E. McLelland, and D. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, pages 90–105, October 1998. 3.1.1, 3.2.1.2

[Kra94] Andreas Krall. Improving semi-static branch prediction by code replication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 97–106, Orlando, Florida, June 20–24 1994. 6.1, 6.3

[KRS94a] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994. 1.1.1, 2.3.3, 2.3.3.1, 5.8, 6.4.1, 6.6

[KRS94b] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, Orlando, Florida, June 20–24 1994. 2.3.3

[Lar99] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, Georgia, May 1–4 1999. 2.3.2.1

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 24(2):197–218, February 1994. 2.4

[LCA01] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, Arizona, April 4–6 2001. 3.2.2

[LCH+03] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 289–299, San Diego, California, June 9–11 2003. 1.1.2

[LCK+98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, Montreal, Canada, June 17–19 1998. 2.3.3, 2.3.3.2, 5.7.1, 5.8

[LCM97] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 4–13, Research Triangle Park, North Carolina, December 1–3 1997. 1.1, 6.1

[LDAS04] Cullen Linn, Saumya K. Debray, Gregory Andrews, and Benjamin Schwarz. Stack analysis of x86 executables. Technical Report TR04-16, Department of Computer Science, University of Arizona, 2004. 2.4.2

[LE95] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 151–162, La Jolla, California, June 18–21 1995. 2.3.1.6

[LFK+93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993. 2.3.1.6

[LMP+04] Chi-Keung Luk, R. Muth, H. Patil, R. Cohn, and P. Geoffrey Lowney. Ispike: A post-link optimizer for the Intel Itanium architecture. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 15–26, San Jose, California, March 20–24 2004. 1.1.2, 2.4.2

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, California, June 18–21 1995. 2.4.2

[LS00] Thierry Lafage and Andre Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload characterization of emerging computer applications*, pages 145–163, Austin, Texas, September 16 2000. 3.3

[MAA$^+$02] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Performance simulation tools. *Computer*, 35(2):38–39, February 2002. 3.2.2

[Mah92] Scott Alan Mahlke. Design and implementation of a portable global code optimizer. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, December 1992. 2.1.3, 3.1.1

[MCE00] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, Monterey, California, December 10–13 2000. 2.3.2

[MCE$^+$02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. SimICS: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. 3.2.2

[McF89] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, Massachusetts, April 3–6 1989. 2.3.1.5

[McF91a] Scott McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 71–79, Toronto, Canada, June 24–28 1991. 2.3.1.4, 2.3.1.4, 3.2.1.2

[McF91b] Scott McFarling. *Program Analysis and Optimization for Machines with Instruction Cache.* PhD thesis, Computer Systems Laboratory, Stanford University, 1991. 2.3.1.5

[MCG$^+$92] Scott A. Mahlke, William Y. Chen, John C. Gyllenhaal, Wen-Mei W. Hwu, Pohua P. Chang, and Tokuzo Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 808–817, Minneapolis, Minnesota, November 16–20 1992. 2.3

[MDWdB01] Robert Muth, Saumya K. Debray, Scott Watterson, and Koen de Bosschere. Alto: A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31(6):67–101, January 2001. 1.1.2, 2.3.1.1, 2.3.1.4, 2.4.2, 3.1.1, 3.2.1, 3.2.1.1, 3.2.1.2, 3.3.1.3

[ME02] D. Mosberger and S. Eranian. *IA-64 Linux Kernel Design and Implementation*, Chapter 9.3: Kernel Support for Performance Monitoring. Hewlett-Packard, 2002. 2.4.2

[MM97] Mayan Moudgill and Jaime H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. Technical Report RC-20857, IBM Research Report, May 1997. 4.8, 4.3.3, 4.4, 7.4

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 98(9), April 1965. 1.1

[Mor98] Robert Morgan. *Building an Optimizing Compiler.* Digital Press, 1998. 2.1.3, 3.1.1

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979. 1.1.1, 2.3.3, 2.3.3.1, 5.8, 6.6

[MR94] Uma Mahadevan and Sridhar Ramakrishnan. Instruction scheduling over regions: A framework for scheduling across basic blocks. In *Proceedings of the 1994 SIGPLAN Symposium on Compiler Construction*, pages 419–434, Edinburgh, Scotland, June 23–25 1994. 2.3.1.6

[MRS$^+$01] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, November 2001. 2.2.4, 6.2.1, 6.6, 7.4

[Muc97a] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 21: Case Studies of Compilers and Future Trends, pages 705–746. In [Muc97d], 1997. 1.1, 6.2.2

[Muc97b] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 13: Redundancy Elimination, pages 377–424. In [Muc97d], 1997. 1.1.1

[Muc97c] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 10: Alias Analysis, pages 293–318. In [Muc97d], 1997. 1.1.1, 4.3

[Muc97d] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, 1997. 2.1, 7.4

[Muc97e] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 4: Intermediate Representations, pages 67–104. In [Muc97d], 1997. 2.1.3, 3.1.1

[Muc97f] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 8: Data-Flow Analysis, pages 217–266. In [Muc97d], 1997. 1, 2.2.1, 2.2.2, 4.2.1, 4.2.2, 2, 5.3

[Muc97g] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 9: Dependence Analysis and Dependence Graph, pages 267–292. In [Muc97d], 1997. 2, 2.2.3.1

[Muc97h] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 16: Register Allocation, pages 481–530. In [Muc97d], 1997. 2.2.1, 2.4.2

[Muc97i] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 1: Introduction to Advanced Topics, pages 1–18. In [Muc97d], 1997. 2.3

[Muc97j] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 17: Code Scheduling, pages 531–578. In [Muc97d], 1997. 2.3.1.3, 2.3.3.3, 3.2.1.2, 6.6

[Muc97k] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 14: Loop Optimizations, pages 425–460. In [Muc97d], 1997. 2.3.1.4, 2.3.3

[Muc97l] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 12: Early Optimizations, pages 329–376. In [Muc97d], 1997. 2.3.3, 3.2.1.2

[Muc97m] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Chapter 18: Control-Flow and Low-Level Optimizations, pages 579–606. In [Muc97d], 1997. 5.6.2

[Mut98] Robert Muth. Register liveness analysis of executable code. Technical Report TR98-16, Department of Computer Science, University of Arizona, 1998. 2.2.1, 2.4.2

[Mut99] Robert Muth. *Alto: A Platform for Object Code Modification*. PhD thesis, Department of Computer Science, University of Arizona, 1999. 2.3.1.1, 2.3.1.3, 2.4.2, 3.2.1.2, 3.3.1.3

[MW92] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 332–330, San Francisco, California, June 15–19 1992. 2.3.3, 6.4.1, 6.5, 6.6

[MW95] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 56–66, La Jolla, California, June 18–21 1995. 2.3.3, 2.3.3.3, 6.1, 6.3, 6.6

[MWD00] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code specialization based on value profiling. In *Proceedings of the 7th International Symposium on Static Analysis*, pages 340–359, Santa Barbara, California, June 29–July 1 2000. 2.3.2.1, 2.4.2

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, New York NY, 1999. 2.2, 2.2.3.2, 2

[NS01] Sebastien Nussbaum and James E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Barcelona, Spain, September 8–12 2001. 3.3

[OG98] Soner Onder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 80–89, Chicago, Illinois, May 14–16 1998. 3.2.2

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 67–78, La Jolla, California, June 18–21 1995. 2.2.4, 6.1, 6.2.1, 6.3, 6.3.2, 6.6, 7.4

[PGM00a] M. A. Postiff, D. A. Greene, and T. N. Mudge. The store-load address table and speculative register promotion. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–244, Monterey, California, December 10–13 2000. 4.3.3, 4.4, 7.4

[PGM00b] Matthew Postiff, David Greene, and Trevor Mudge. The need for large register files in integer codes. Technical Report CSE-TR-434-00, EECS/CSE University of Michigan, 2000. 2.3.3.2, 5.4.2, 5.8

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, June 1990. 1, 1.1.2, 2.3.1.5, 3.2.1.1, 4.4

[Rau91] B. R. Rau. Data-flow and dependence analysis for instruction level parallelism. In *Proceedings of the 4th International Workshop on Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 236–250. Springer-Verlag, 1991. 2.2.4, 6.6

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997. 3.2.2

[RBG$^+$01] Alex Ramírez, Luiz Barroso, Kourosh Gharachorloo, Robert Cohn, Josep L. Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 155–164, Göteborg, Sweden, June 30–July 4 2001. 2.4.2

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–34, Paris, France, December 2–4 1996. 1

[RCT$^+$98] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided load marking for memory renaming. Technical Report UCSD-CS98-593, University of California, San Diego, 1998. 2.3.2.1, 4.4

[Rei95] Jeff Reilly. SPEC describes SPEC95 products and benchmarks. *SPEC Newsletter*, 7(3), September 1995. 3.3

[RF93] B. R. Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, May 1993. 2.3.1.6

[RG89] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989. 2.3.1.4

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Program-*

ming Languages and Operating Systems, pages 38–50, Boston, Massachusetts, October 12–15 1992. 2.3.1.6

[RLPN$^+$99] Alex Ramírez, Josep L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 119–126, Rhodes, Greece, June 20–25 1999. 1, 2.3.1.5

[RLPV01] Alex Ramírez, Josep L. Larriba-Pey, and Mateo Valero. Instruction fetch architectures and code layout optimizations. *Proceedings of the IEEE*, 89(11):1588–1609, November 2001. 2.3.1.5

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–252, Toulouse, France, September 6–10 1999. 4.3, 4.4

[RVL$^+$97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows/NT Workshop*, pages 1–7, Seattle, Washington, August 11–13 1997. 1.1.2, 2.4.2

[Sar89] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 298–312, Portland, Oregon, June 19–23 1989. 1.1.2, 2.3.2.1

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver, Canada, June 18–21 2000. 2.2.4, 6.2.1, 6.6, 7.4

[Sch77] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977. 2.3.1.4

[SCK$^+$93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993. 2.4

[SDA02] Noah Snavely, Saumya K. Debray, and Gregory Andrews. Predicate analysis and if-conversion in an Itanium link-time optimizer. In *Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, November 2002. 1.1.2, 2.4.2

[SDA03a] Noah Snavely, Saumya K. Debray, and Gregory Andrews. Unscheduling, unpredication, unspeculation: Reverse engineering Itanium executables. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 4–13, November 2003. 2.4.2

[SDA03b] Noah Snavely, Saumya K. Debray, and Gregory Andrews. Unspeculation. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 205–214, October 2003. 2.4.2

[SDAL01] Benjamin Schwarz, Saumya K. Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the 3rd Workshop on Binary Translation*, Barcelona, Spain, October 2001. 1.1.2, 2.3.1.4, 2.4.2

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, Florida, June 20–24 1994. 2.4, 2.4.2, 3.1.1

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 15–17 1997. 2.2.3, 4.1, 4.4

[SHK04] Bernhard Scholz, Nigel Horspool, and Jens Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 221–230, Washington, D. C., June 11–13 2004. 2.3.3.1, 5.8, 6.6

[Sil99] Silicon Graphics, Inc. *MIPSpro Compiling and Performance Tunning Guide*. Number 007-2360-008. Mountain View, CA, 1999. 1.1.2

[SL98] Eric Schnarr and James. R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, San Jose, California, October 2–7 1998. 3.2.2

[SLM96] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, Philadelphia, Pennsylvania, May 22–24 1996. 6.1, 6.3

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, May 12–14 1981. 1.1, 6.1

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982. 1.1, 5.1

[Smi91] Michael D. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, 1991. 2.3.2.1, 2.4.2, 3.1.1, 3.1.2.2, 3.2.1.2, 2

[SP89] J. J. Shieh and C. A. Papachristou. On reordering instruction streams for pipelined processors. In *Proceedings of the 22th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–206, Dublin, Ireland, August 14–16 1989. 2.3.1.6

[SPC01] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Barcelona, Spain, September 8–12 2001. 3.3

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, San Jose, California, October 5–9 2002. 3.3

[SS95] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12), December 1995. 1.1

[SW92] 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. 1.1.2, 2.2.1, 2.3.1.1, 2.4.2, 3.1.2.1

[SW94] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN*

1994 Conference on Programming Language Design and Implementation, pages 49–60, Orlando, Florida, June 20–24 1994. 2.4.2

[TM01] Arial Tamches and Barton P. Miller. Dynamic kernel code optimization. In *Proceedings of the 3rd Workshop on Binary Translation*, Barcelona, Spain, October 2001. 3.2.1.2

[TP95] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 3–7 1995. 2.2.4, 6.6

[TXD95] Josep Torellas, Chun Xia, and Russell Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 360–369, Raleigh, North Carolina, January 22–25 1995. 2.3.1.5

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 264–275, Palo Alto, California, July 25–27 1986. 1.1.2

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, California, April 8–11 1991. 2

[Wal92] David R. Wallace. Cross-block scheduling using the extended dependence graph. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 72–81, Washington, D. C., July 19–24 1992. 2.3.1.6

[Wei97] Reinhold Weicker. On the use of SPEC benchmarks in computer architecture research. *Computer Architecture News*, 25(1):19–22, March 1997. 3.3

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 18–21 1995. 2.2.3, 4.1, 4.4

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, North Carolina, December 5–8 1993. 2.4

[WM95] W. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995. 1.1

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991. 2.3.1.1

[YGS95] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, S. Margherita Ligure, Italy, June 22–24 1995. 6.1, 6.3

[YP92] Tse-Yu Yeh and Yale Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, Queensland, Australia, May 19–21 1992. 1.1, 6.1

[ZG01] Youtao Zhang and Rajiv Gupta. Timestamped whole program path representation. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 180–190, Snowbird, Utah, June 20–22 2001. 2.3.2.1

[Zha02] Youtao Zhang. *The Design and Implementation of Compression Techniques for Profile Guided Compilation*. PhD thesis, Department of Computer Science, University of Texas, 2002. 5.2.2