ALTO: A PLATFORM FOR OBJECT CODE MODIFICATION

by

Robert Muth

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 9

ALTO: A PLATFORM FOR OBJECT CODE MODIFICATION

Robert Muth, Ph. D.

The University of Arizona, 1999

Director: Saumya K. Debray

This dissertation describes `Alto`, a platform for object code modification for Digital Unix/Alpha executables. Object code modification, also called binary rewriting, allows us to change compiled and linked programs, thereby extending the process of code generation well past the compilation phase of a program.

Object code modification is becoming increasingly important. One reason for this is the recent trend of making programs available as executables only — without the corresponding source code.

We explain the difficulties encountered by object modification, especially in the area of program analysis, and show how they are dealt with in `Alto`. Several improvements to register liveness analysis are presented.

`Alto` has been used to implement an optimizer which allows us to evaluate the benefits of classical compiler optimizations when applied to object code. This optimizer outperforms the vendor-supplied optimization tools significantly.

`Alto` has also been used to instrument programs in order to generate sophisticated execution profiles, such as value profiles. We show how such profiles can be profitably exploited using a novel technique — guarded code specialization — and how this optimization can be incorporated into the optimizer.

Finally, we consider the issue of code compression, i.e., using `Alto` to make programs smaller rather than to make them faster. A variety of transformations are presented which are able to reduce the code size of programs substantially.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowed without special permission. provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted bt the copyright holder.

SIGNED: _____

ACKNOWLEDGEMENTS

Credit for much of the work described in this dissertation belongs to my tireless advisor, Professor Saumya Debray. He provided for an excellent research environment, left me enough freedom to do things the way I thought they should be done, and was always available to discuss ideas and problems.

I would also like to thank my committee members Peter J. Downey and William Evans for numerous helpful discussions and for their patients with revising early drafts of this document.

Other people that have worked on `Alto` also deserve my gratitude: Koen De Bosschere developed an early prototype of `Alto`. Bjorn De Sutter and Scott Watterson worked with me on the current version.

Last but not least I would like to thank my parents Guenther and Yvonne. They made the right decisions for my education when I was young, and later on provided moral support and encouragement when it was up to me to make decisions.

This work was supported in part by the National Science Foundation under grants CCR-9502826 and CCR-9711166.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This dissertation describes `Alto`, a platform for object code modification for Digital Unix/Alpha executables. Object code modification, also called binary rewriting, allows us to change compiled and linked programs, thereby extending the process of code generation well past the compilation phase of a program.

Object code modification is becoming increasingly important. One reason for this is the recent trend of making programs available as executables only — without the corresponding source code.

We explain the difficulties encountered by object modification, especially in the area of program analysis, and show how they are dealt with in `Alto`. Several improvements to register liveness analysis are presented.

`Alto` has been used to implement an optimizer which allows us to evaluate the benefits of classical compiler optimizations when applied to object code. This optimizer outperforms the vendor-supplied optimization tools significantly.

`Alto` has also been used to instrument programs in order to generate sophisticated execution profiles, such as value profiles. We show how such profiles can be profitably exploited using a novel technique — guarded code specialization — and how this optimization can be incorporated into the optimizer.

Finally, we consider the issue of code compression, i.e., using `Alto` to make programs smaller rather than to make them faster. A variety of transformations are presented which are able to reduce the code size of programs substantially.

CHAPTER 1

INTRODUCTION

1.1 Motivation

This dissertation is about direct modifications of object code. Such modifications may occur either at a very late stage during linking (link time) or after linking (post link time). Both approaches are quite similar: integrating modifications within the linker will simplify parsing of the code and might give access to slightly more information about the code, while changing object code after linking provides a very clean separation of responsibilities and does not require access to potentially proprietary linker source. In what follows we will not distinguish between the two approaches.

Traditionally, it is the task of the compiler or assembler to generate object code and it seems complex and cumbersome to change object code once it has been produced. Nevertheless, the number of applications where object code modification is successfully employed grows rapidly. This is partly due to the fact that computers are becoming powerful enough to cope with the often quite high resource demands of object code modification.

The following list describes the most popular applications of object code modification.

Customization Most software vendors ship software in executable form. Because of the high maintenance and testing cost there is a reluctance to produce more than one version of an executable for any one platform. To ensure that the software works

on all systems, the vendors aim their executables at the lowest common denominator of the architecture. However, the systems the software runs on might be quite different, their CPUs might have slightly different instruction sets, cache sizes, pipelines, functional units, even the number of CPUs might be different. Consider, for example, the Windows 95 operating system which runs on such different CPUs as: x486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium Celeron, AMD K6-2, AMD K6-3, Cyrix M-2, etc. However, there is only one version of this software available. As another example consider the Alpha family of CPUs which used to lack instructions for loading/storing individual bytes and words. Instead, instruction sequences were used to emulate these elementary operations. Recent members of the CPU family have load and store instructions for bytes. A typical software vendor will most likely compile his programs not using the new instructions to ensure that his software runs on all CPUs. Consequently, users with state of the art systems experience suboptimal performance.

Object code modification can help customizing a program by making use of new features of the CPU when those are present in a system without requiring recompilation or waiting for new compiler releases supporting these features.

Customization usually leads to faster programs. However, one could imagine a situation were the example above is reversed, and in which we are presented with executable code compiled for the latest member the Alpha CPU family. We want to run it on an old version of the CPU, no longer supported by the software vendor. We can use object code modification to replace all the instructions which load and store bytes or words with emulating instructions.

Customization can also be used in combination with profiling to tune a binary for common input data and program usage. Suppose a certain user mostly exercises

the spell checker and the display rendering portion of a word processor. It would be beneficial to reorder those components within the program to reduce the likelihood that they conflict in the instruction cache. Furthermore, assume that this person almost exclusively uses the Times font in his word processor documents. Clearly, this will change the behavior of the display rendering code: certain branches will or will not be taken with higher probability and certain values will be more likely to populate certain registers. Adapting the display rendering code for this common case — possibly at the expense of slowdowns for the uncommon case (e.g., the Helvetica font) — might be greatly beneficial to this particular user.

Binary Translation Taking customization to the extreme, we can attempt to translate the object code to run on a different platform. A few such translators have been implemented commercially: `FX!32` is a Windows NT/x86 to Windows NT/Alpha translator [15] , Freeport Express is a Solaris/Sparc to Digital Unix/Alpha translator [71], `VEST` translates OpenVMS/VAX to OpenVMS/Alpha [70, 64], and `mx` translates Ultrix/Mips to Digital Unix/Alpha [70, 64].

Binary translation is also useful for the fast emulation of a new or fictitious platform before actual hardware is available, allowing compiler writers, for example, to test their code generator in advance.

Binary translation often requires a runtime software emulator for the source platform, in order to cope with code that is generated on the fly and which cannot be statically translated. A translation that falls back to an emulator for the source platform is called hybrid translation. The above mentioned FX!32 system actually uses emulation by default and will apply binary translation only to the frequently executed portions of a program in a separate offline step.

Translators can obviate the need for porting software if one is willing to pay the

price of a small performance penalty.

Program Analysis/Profiling Program analysis and profiling tools are the most popular applications for object code modification. Tools that instrument programs to determine basic block execution frequencies are very common. This information can be used by the compiler for profile-driven optimizations, by developers to help them focus their tuning efforts on the relevant parts of the code. Architects might use profiling to determine the dynamic instruction mix of applications or their data and instruction cache behavior [75, 47]. Other tools instrument the code to examine the accuracy of branch predictions or scheduling decisions made by the compiler without the use of special hardware (such as bus monitoring systems) or simulation. A common class of tools instrument object code to obtain address traces, which help architects improve cache design. These traces are very large, often consuming several gigabytes of disk space. Instead of writing them to disk and processing them offline, a recent trend adds the processing code to the object code and invokes it whenever a new piece of trace information would normally be written to disk. While the first approach typically causes the instrumented program to run about 100 times slower [49], the second approach reduces this slowdowns to a factor of 10 [66].

There are, of course, other ways of obtaining profiling information. One is to make the compiler instrument the code [38], and another is to use statistical methods [4]. Instrumenting code at a higher level than object code may not yield accurate information of the type computer architects care about, and it also critically changes the program behavior that we want to analyze (analogously to “Heisenberg’s uncertainty principle” in physics). Statistical methods, on the other hand, sometimes have problems with accuracy.

Debugging Object code modification also has a wide range of useful applications in debugging. Suppose that because of a programming error a memory cell is accidentally overwritten. Locating the point where the overwrite occurs can be a very difficult task. If there is no hardware support on the CPU we could resort to modifying the compiler to add checks, but this would be slow and libraries would not be covered. Another approach which has been used in the past and which is also rather slow is to single step through the program, continuously checking whether the memory cell has changed. A more efficient solution is to modify the object code to add checking code before each write instruction that will trap on a write to the memory cell in question.

Purify uses object code modification to detect memory leaks, out of bounds memory/array accesses, and use of uninitialized data [40].

`JiTI`, A debugging aid with a somewhat different flavor, inserts instrumentation code before load and store instructions in a parallel program which will assure that these loads and stores occur in a formerly observed order. This enables deterministic replays in (shared memory) parallel environments which is a great help in reproducing bugs [62, 61].

Software Fault Isolation (Sandboxing) Software fault isolation is closely related to the debugging techniques mentioned above. We describe a scenario from [72]: Suppose we have a piece of untrusted object code which we want to link with trusted code. One concern is that the piece of untrusted code might accidentally modify data structures maintained by the trusted code and hence corrupt the system. To address this problem we could assign the untrusted code to its own segment within the applications address space, and add checking code before each read/write instruction. The checking code will trap on a read/write attempt outside of the segment.

The overall effect is very similar to that which Java achieves with a combination of type checking and runtime checks.

Another application is the prevention of certain security attacks that exploit buffer overflows on the stack. Most attacks exploit buffer overflows inside the programs to (a) create a subroutine that spawns a root shell on the stack (b) overwrite the return address in the current stack frame with the start address of the newly created subroutine. We can prevent this attack by inserting checks before each indirect control transfer (`jump,return`), to validate that the address being jumped to lies in a valid range.

Code Optimization Intuitively, code optimization should be the domain of the compiler, since it has access to high level information such as data types, control structures, alias information, etc., which greatly aids in generating efficient code and which is not readily available at the object code level. So why bother optimizing object code?

- We want to be compiler/language independent.

  Working with object code makes our optimizations essentially compiler and language independent, similar to a common back end used with several front ends. However, we still may need to recognize certain compiler and/or language specific idioms at the object code level (like computed jumps) and treat them specially, in order to improve the effectiveness of the optimizations.

- We want to add a new optimization to a compiler.

  Often we do not have access to the compiler source. Or, the documentation of the compiler source is so poor that adding the new optimization might be difficult. Hence, it is very popular to try out new optimizations in a simple and

well documented compiler like `lcc` [34] 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, on the other hand, allows us to essentially add optimizations to the best available compiler without modifying it.

- The program source or parts thereof (libraries) are unavailable.
  Especially for old programs (legacy software), source code is often unavailable or it is unclear which version of the source corresponds to the program we want to optimize. Optimizing the object code appears to be the only way to improve performance of these programs.

- The optimization cannot be easily performed at compile time.
  Consider the case where we want to improve the control transfer (jump) code for subroutine invocations. Depending on the distance of the jump (in number of bytes), different jump instructions can be chosen, a pc-relative jump with short displacement, a pc-relative jump with long displacement, or an absolute jump. Unfortunately, the compiler is usually not able to estimate the jump distance, and hence needs to pick the most conservative and hence suboptimal instruction, viz. the absolute jump. At link time, on the other hand, we know exactly what the jump distance is and can pick the optimal instruction.

- We want to perform whole program optimization.
  Whole program optimization can, in theory, be done at compile time but this is often hindered by missing source for library code. This problem does not exist for statically linked object code.

- We want to utilize profiling information obtained at the object code level.
  Generating profiling information by instrumenting object code is very popu-

lar and, because of tools like `Atom` (cf. 1.2), also fairly easy. 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. This "impedance mismatch" problem is also found in a source level debugger. Such a debugger actually works at the object code level, but needs to back map the information to source code. This is a hard problem — especially when the code is highly optimized. When optimizing at the object code level, on the other hand, the mapping is one-to-one and does not present any problems.

Code Compression/Compaction While the cost metric we tried to reduce in the previous cases was time, we may also be concerned about space. Besides classical optimizations which usually also reduce code size, we can reduce code size using special compression techniques. Compressed code must either be decompressed before execution (called wire representation) [31] or it can be executed without decompression [35, 21]. The first method results in a smaller compressed representation than the second, but requires the overhead of decompression before execution. This overhead may be negligible and in fact maybe compensated for by the savings in transmission or retrieval cost. A more severe problem, however, is that it requires space for the decompressed code.

The second method preserves executability of the code and is therefore more amenable to object code modification even though the borders between the two methods are somewhat flowing. If we prepend a piece of code to the wire representation that first performs decompression and then runs the decompressed executable, we have technically preserved executability. Tools like this were very popular when computers were not equipped with hard disk drives and one tried to

cram as much information as possible onto floppy disks [59].

It is also possible to add some sort of interpretive techniques to an executable to reduce space requirements. For example, on the Motorola 68000 based Atari ST computer the designers were not able to fit the entire operating system into the 192 kB ROM. So they replaced common opcode sequences with illegal instructions [1] and installed an interpretor to handle the illegal instruction exceptions [42].

A similar but less system specific mechanism factors common code sequences into subroutine calls [35, 27, 21]. This can in principle be done by a compiler but often the intermediate representations used in the compiler do not provide enough support for this kind of transformation. In addition, more of the code is visible at link time, e.g., libraries, increasing the number of opportunities for factoring.

This dissertation describes `Alto` (A Link Time Optimizer), a platform for modification of object code. `Alto` has been implemented for Digital Unix/Alpha executables and is being ported to Linux/Alpha. The main emphasis of the dissertation will be on object modification for optimization, code compression, and profiling.

Despite being a fairly system specific piece of software, the experience gained with `Alto` should be transferable to other platforms/architectures (especially RISC based systems) since the Alpha is a very generic RISC CPU. In fact, its instruction set is very similar to low level code representations such as ILOC [58], LIR [55], and Omnicode [73, 1] so that `Alto` could also be viewed as a backend for monolithic compilation.

[1] The illegal instructions were taken from a pool of reserved opcodes called “line-f” because the first hexadecimal digit of each opcode was “f”. Later those opcodes became legal floating point instructions and the scheme was abandoned.

1.2 Related Work

This section describes other projects in the area of object code modification and points out their differences from `Alto`.

OM

`OM` is an optimizer for executables initially implemented for DEC-stations running Ultrix/Mips, and later ported to Digital Unix/Alpha.

`OM` was designed as a separate pass after linking but, unlike `Alto` relies on the linker to provide additional information not found in the executable. `OM` can also make use of profiling information [67, 68].

One of the design goals for `OM` has been to make it fairly light-weight. Compared to `Alto`, it does not perform many optimizations, and the ones it does perform are restricted to those that do not consume a lot of resources. The following is a list of optimizations performed by `OM`:

- Code size reduction by unreachable code removal
- Compaction of the memory area that holds compile time constants by elimination of unused and duplicate constants
- Reordering of global data structures (variables) to provide more efficient access
- Profile guided code positioning and alignment
- Instruction (re-)scheduling
- Peephole optimization
- User-directed procedure inlining

ATOM

`Atom` (Analysis Tool with `OM`) is an instrumentation tool generator for the Digital Unix/Alpha platform which originated in the `OM` project but has since then diverged. `Atom` is a very mature and user-friendly tool which is used extensively inside DEC (now COMPAQ). It has been used for many big applications including OS kernels [66].

`Atom` strictly separates the tool specific part from the common infrastructure needed by all tools. The tool specific part consists of an analysis component and an instrumentation component. Both components can be written entirely in a high level language (typically C), which distinguishes `Atom` from any other tool, including `Alto`.

The instrumentation code is linked with the `Atom` instrumentation engine to create an instrumentation tool. This tool will parse an executable and insert function calls at specific places via the `Atom` API. The functions called are those defined in the analysis code. Calls to these functions can be inserted before/after program execution, shared library loading, procedures, basic blocks, or instructions. The parameters passed to the functions are determined by the instrumentation code. Possible parameters are: current register values, instruction fields, symbol names, addresses, etc.

`Atom` even allows the analysis code to dynamically allocate memory. This is non-trivial because memory allocated by the analysis code should not be visible to the instrumented program in order to preserve program behavior as much as possible, i.e., the values returned by calls to `malloc()` should be the same in the original and in the instrumented version of the program.

Registers modified by an analysis routine are saved to the stack and later restored. Some attempts are made to reduce this overhead but it is still quite significant. Programs instrumented using `Atom`/`pixie` (see below) typically suffer a slowdown of a factor of 2 to 3.

Among the tools that have been (re-)implemented with `Atom` are:

- `pixie`. A reimplementation of a basic block execution frequency profiling tool. The profile generated by pixie is used by `OM` to guide some of its optimizations [69, 75].
- `Third Degree`. A memory leak detection tool.
- `Hiprof`. A performance analysis tool that collects data similar to, but more accurate than, `gprof`.

SPIKE

`Spike` is an adaptation of `OM` to the Windows NT/Alpha platform. `Spike` consists of an instrumentation part and optimization part. Both are embedded in the `Spike` optimization environment (SOE), which transparently handles the task of collecting and managing profiling information for the user [18, 17, 36].

`Spike` is aimed at call intensive programs, with loops that span multiple procedures and procedures that have complex control flow and contain numerous basic blocks.

The instrumentation part is a `pixie` adaptation which provides basic block and control flow edge execution frequency counts. A minimum of basic blocks and edges are instrumented and register liveness analysis is used to find free scratch registers for the instrumentation code. The instrumentation code bloat is thereby only 30%. There are plans to replace the instrumentation part by statistical sampling using DCPI [4].

`Spike` automatically scans the executable for the dynamically linked libraries (DLLs) it uses and processes them as well.

The important transformations performed by the optimization part are profile-driven:

- Pettis-Hansen style profile guided code placement [57] improves instruction cache performance.

- Hot cold optimization (HCO) reduces the length of the most frequently executed paths in a procedure.

`Spike` reportedly speeds up program execution by as much as 33%, which seems to be mostly due to the profile guided code placement. The HCO optimization benefit is unclear since no execution time improvements are reported. `Spike` seems to be most effective with call intensive programs. Programs that spend a significant amount of time in inner loops, e.g., FORTRAN programs, usually get very little speedup. Compared to `Alto` very few optimizations have been implemented.

EEL

`EEL` (Executable Editing Library) is a C++ library that tries to hide much of the complexity and system specific detail of editing executables. It was developed at the University of Wisconsin-Madison and runs on Solaris/Sparc and Ultrix/Mips. [48]

`EEL` tries to be as system and machine independent as possible. Theoretically, tool builders should be able to modify an executable without being aware of the details of the underlying architecture or operating system, or being concerned with the consequences of deleting instructions or adding foreign code. `EEL`'s programming interface is not as high level as `Atom`'s but the programmer has more control over the instrumentation process since `Atom` can only insert subroutine calls and not modify existing instructions. As an intermediate representation `EEL` employs a machine independent RISC-like abstract instruction set. However, instrumentation code (called snippets) consists of concrete instructions that must be rewritten in machine language for different architectures. A register scavenging scheme that takes advantage of calling conventions is used to provide the scratch registers for the snippets, and to reduce the amount of register spilling. Hence the snippet defined by the tool writer and the actual code added as instrumentation

to an executable might differ in the allocation of registers. If other changes are wanted, e.g., adjustments of offset/displacements, the tool writer needs to back-patch the snippet just before it is added to the binary. The number of instructions is not allowed to change at this point.

`EEL` does not use relocation information and falls back to runtime code when static analysis is insufficient. This also prevents the user from editing certain basic blocks (typically 15-20% of all basic blocks) which are excluded from instrumentation. The authors claim that in most case alternative basic blocks can found and edited instead. For this reason `EEL` is a instrumentation platform rather than optimization platform.

`EEL` has been used to reimplement the `qp` and `qpt` tools [8] which are used to obtain path profiles.

Etch

`Etch` is a binary modification tool for Windows NT/x86 executables. It was developed jointly at the University of Washington and Harvard University and its architecture was strongly influenced by `Atom`. Like `Atom` it separates instrumentation from analysis. To instrument a program, `Etch` is invoked with the name of an executable and a dynamically linked library (DLL). The DLL contains the analysis code in the form of callback functions that are invoked by `Etch` to modify the executable. Those functions can in turn call the `Etch` API to perform the actual instrumentation [60].

`Etch` includes a runtime library with the modified executable which might make it less suitable for optimizations. The only reported optimization is code layout based on the Pettis-Hansen algorithm [57]. Like `Spike` it can handle dynamically linked executables and will instrument/optimize DLLs used by a program.

1.3 Contributions of Alto

`Alto` has been implemented in the C programming language and works reliably on all programs tested. The software can be downloaded free of charge from the `Alto` webpage **http://www.cs.arizona.edu/alto/**.

The main contributions are listed below.

1. Program analysis (Section 3)

   We present several improvements to register liveness analysis. We show how to preserve correctness of the analysis in the presence of control flow anomalies typically not encountered in high level languages but frequently observed in object code. A novel insight about the fixpoint equations for the analysis is exploited to speed up its computation time by 25%. We also show how the consideration of calling conventions and callee saved registers decreases the number of live registers. Furthermore, We examine space-time tradeoffs and space/time-precision tradeoffs.

2. Classical compiler optimizations (Chapter 4)

   We evaluate the usefulness of an extensive set of classical compiler optimizations in the context of link time code modification. Common sense suggests doing classical compiler optimization in the compiler. However, we find significant optimization opportunities at link time. Programs optimized with our system typically run 6% faster than those produced with the vendor-supplied compiler infrastructure alone.

3. Common case specialization (Chapter 5)

   We discuss guarded code specialization, a new optimization based on value profiles, which could also be incorporated into ordinary compilers. We show how to

select program points for value profiling and present a cost-benefit analysis to automatically determine which of those program points are to be specialized for what value. Guarded code specialization results in an additional speedup of up to 10% for some programs.

4. Code compression (Chapter 6))

   We examine opportunities for code size reduction via object code modification. Using code factoring transformations on top of classical compiler optimizations, we are able to reduce code size by 38% on the average.

CHAPTER 2

OVERVIEW OF THE ALTO SYSTEM

Object code modification is a three phase process consisting of :

- Parsing — transformation of the object code into an intermediate representation
- Editing — manipulation of the intermediate representation
- Code Generation — transformation of the intermediate representation back into object code

In this chapter we will discuss these phases, describe the common problems encountered, and show how `Alto` handles them.

We assume a generic Unix executable format depicted in Figure 2.1 [39]. Besides the file representation, the runtime organization in memory is also shown.

The Program Header contains offsets and sizes of the segments and tables and their location in the address space. It also contains the code address where execution starts. The Text Segment contains read-only data, i.e., the program code and constants. The Data Segment contains initialized data that is read/writable. The BSS Segment contains zero initialized read/writable data and is therefore reduced to an address/size pair in the file representation. The Relocation Table contains information which allows us to change the program to run at different absolute positions in the address space. The Symbol Table contains information traditionally used by a debugger to establish a correspondence

between the source code and the object code derived from it. The symbol table is optional and can be removed from the executable using the `strip` command. If present, `Alto` will use it to make its output more user-friendly. For example, instead of reporting that "the subroutine at address 0x120003ac has been inlined", the message "subroutine `memcpy` has been inlined" will be printed.

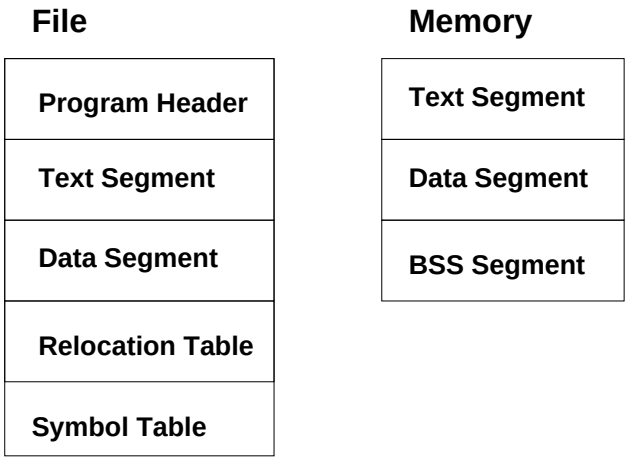


Figure 2.1: Generic executable format

2.1 Parsing

The task of parsing is to transform the object code into some intermediate form which is more suitable for further modifications.

`Alto` is using a three-address intermediate language that is very close to the Alpha machine language [2]. Although this might seem to make `Alto` very non-portable, this three address code is in fact very similar to the low level intermediate representation used in many compilers [54, 55, 58].

2.1.1 Code Discovery

Code discovery tries to locate the parts of the object file that contain executable instructions. Locating code is not always straightforward. Often read-only data (string constants, jump tables, floating point constants) and program code are interleaved in the Text Segment. This problem is aggravated if the architecture has variable length instructions (like the Intel x86 architecture). In this case one has to be very careful where to start decoding instructions. Usually one has to couple code discovery with control flow graph construction, i.e., whenever a new branch target is identified one starts decoding instructions at that address until one encounters a control flow changing instruction. In the presence of indirect jumps, however, it might still be impossible to discover all code.

Fortunately, on the Digital Unix/Alpha platform, compilers are very disciplined and place code and data into separate areas of the Text Segment. Furthermore all instructions are 32 bits wide.

If the program is dynamically linked, code discovery may also try to identify the shared libraries used by the program and to parse them as well. `Alto` currently does not support dynamically linked code.

2.1.2 Control Flow Graph Construction

An important part of the intermediate representation is the control flow graph, which is also essential for the dataflow analyses performed during subsequent phases.

Conceptually, an executable consists of a set of subroutines (functions) denoted as *Functions*. The distinguished subroutine *entryfun* designates where the execution of the program begins. Each subroutine f consists of a collection of nodes (basic blocks) *Nodes*$[f]$. A node *n* consists of a sequence of instructions *Instructions*$[n]$, in which control always enters at the beginning and leaves at the end without intervening branches.

The first instruction of a node is also called *leader*. The collection of all the basic blocks in all subroutines is denoted as *Nodes*. To simplify reasoning about nodes we assign types to them, denoted by $Type[n]$. There is a wide variety of types. The most relevant ones are: *call* for those nodes that initiate a subroutine invocation, *return* for those nodes where execution resumes after the subroutine call, *init* for those nodes starting a subroutine, *exit* for those nodes ending a subroutine. Each subroutine f has exactly one node of type *init* denoted as $InitNode[f]$ and exactly one node of type *exit* denoted as $ExitNode[f]$.

Nodes are connect by directed edges, indicating possible control flow. An edge might connect two nodes within the same subroutine — in which case it is called an intraprocedural edge — or two nodes in different subroutines — in which case it is called an interprocedural edge. The collection of all edges is denoted as *Edges*. The set of immediate successor (resp. predecessor) nodes of a node *n* is denoted $Succ[n]$ (resp. $Pred[n]$).

An interprocedural control flow graph consists of the directed graph created by *Nodes* and *Edges*. It is very similar to the program supergraph described in [56]. The (intraprocedural) control flow graph for subroutine f is the subgraph of the control flow graph induced by $Nodes[f]$.

Creating control flow graphs for programs in high-level languages is straightforward [2]. Matters are somewhat more complex at link time because control flow has been obscured by the compilation process, and because we need to deal with machine-level idioms for control transfer such as computed jumps. The algorithm used by `Alto` to construct a control flow graph for an input program is as follows: As starting points we use all the Text Segment addresses appearing as literal data somewhere in the object code file. Here, relocation information helps us differentiate between real addresses and random bit patterns that just look like addresses. Included with those addresses is the start address of *entryfun* which can be found in the program header.

Now the "standard" algorithm [2] is used to identify more leaders and basic blocks. A leader that is reached by a call instruction begins a subroutine *init* node. [1] A function is assumed to extend from one *init* node until just before the next *init* node, in instruction sequence order. This ensures that each subroutine has exactly one *init* node. The assumption that control enters a subroutine at exactly one point and leaves at exactly one other point is occasionally violated resulting in irregular interprocedural control flow which our analyses and optimizations need to cope with. Section 2.1.4 describes how the introduction of compensation edges can support analyses and optimizations in that case.

Next edges are added to the control flow graph. Subroutine calls are modeled as depicted in Figure 2.2. A *call* edge leads from the basic block containing the call instruction (*call* node) to the target block which is, by definition, a subroutine *init* node. A *link* edge connects the *call* node to the basic block beginning right after the call instruction (*return* node). A return edge leads from the *exit* block of the called function (callee) to the *return* node.

For any call node n_c, $ReturnNode[n_c]$ denotes the corresponding return node and $Callee[n_c]$ denotes the function being called. Similarly, for any *return* node n_r, $CallNode[n_r]$ denotes the corresponding *call* node and $Callee[n_r]$ denotes the function that was called.

[1] Some of the leaders determined by literal addresses also mark function init blocks and this can be determined using the relocation information.

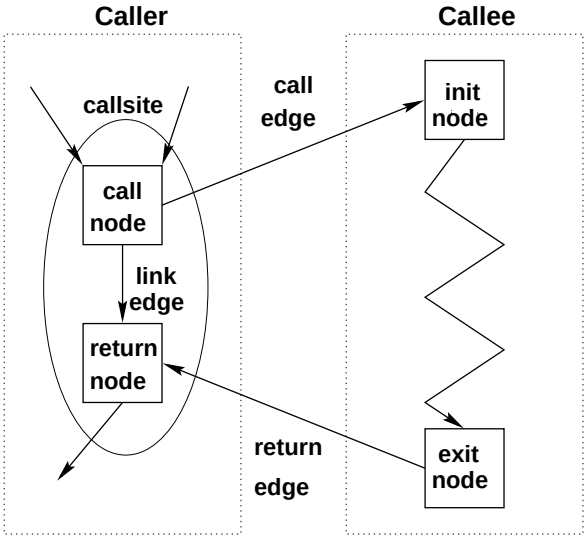


Figure 2.2: Modeling subroutine calls

Unconditional branches are eliminated from the intermediate representation since this information is implicit in the edges.

Whenever an exact determination of the target of a control transfer is not possible, `Alto` estimates the set of possible targets conservatively, using a special node *unknownnode* and a special subroutine *unknownfun*. [2] This simplifies the implementation of data flow analyses because we can associate worst-case data flow assumptions with them and otherwise treat them as ordinary nodes and subroutines. If there is an indirect jump to an unknown target we add an edge from the jump block to *unknownnode* and if there is an indirect call to an unknown subroutine it is simulated by a call to *unknownfun*. Conversely, if the start address of a block appears as literal data somewhere in the object code file, we assume it can be the target of any indirect jump and add an edge from *unknownnode* to the node. If the start address of a subroutine *init* node

[2] *unknownnode* also represents the *init* and *init* node of *unknownfun.*

appears as literal data somewhere in the object code file we assume the subroutine can be the target of an indirect call and simulate a call from *unknownnode*.

2.1.3 Computed Indirect Jumps

`Alto` works hard to find the actual target of indirect jumps and calls and will adapt the control flow graph accordingly. As described in the previous section, a jump with a unknown target will initially be modeled as a jump to *unknownnode*. `Alto` tries to determine whether such a jump is a computed jump derived from a C switch statement (or similar construct in other languages) by pattern matching a code template with the code surrounding the jump. [3] The pattern matching is non-trivial since the compiler might have reordered instructions, peephole-optimized instructions, or moved instructions into different nodes. If we find a match, we have implicitly determined the location and dimension of the jump table and can refine the control flow graph by replacing the edge to *unknownnode* with edges to the actual target nodes.

This transformation is done as part of the editing phase because it greatly benefits from other transformations and analyses, such as liveness analysis.

2.1.4 Control Flow Anomalies

Machine code is not as well behaved as high-level source code. In particular certain assumptions about control flow, which seem reasonable at a higher level, are routinely violated at the machine code level.

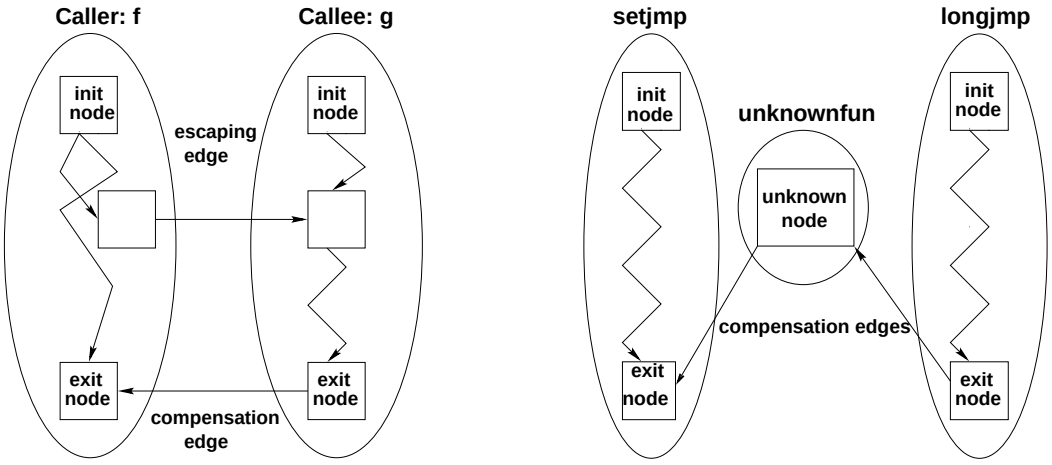
One assumption is that control leaves a subroutine only at its *exit* node or its call sites. At the level of executable code, this assumption can be violated by *escaping branches*, i.e., ordinary (non-subroutine-call) control transfers from one subroutine into another.

[3] The template was derived by inspecting switch statement code produced by various compilers

Typical causes for *escaping branches* are tail call optimization and code sharing in hand-written assembly code (found, for example, in some numerical libraries).

Another assumption is that a subroutine call returns to its caller at the instruction immediately after the call instruction. This assumption is violated by non-local control transfers via subroutines such as `setjmp` and `longjmp`.

`Alto` handles both cases by the inserting additional edges, called *compensation edges*, into the control flow graph as depicted in Figure 2.3.



(a) escaping edge

(b) setjmp/longjmp

Figure 2.3: Use of compensation edges

In the first case, an *escaping branch* from a caller subroutine f to a callee subroutine g results in a single compensation edge from the *exit* node of g to the *exit* node of f. Conceptually, this ensures that control flow entering f can also exit from f — this is important for the data flow analyses to safely approximate program behavior. Without the escape edge, data flow facts cannot be propagated back to f where they originated.

In the second case, the subroutine `setjmp` has a compensation edge from

unknownnode to its *exit* node, while the subroutine `longjmp` has a compensation edge from its *exit* node to *unknownnode*. This models the fact that the location of an invocation of `setjmp` can later be jumped to from unknown places, and that calling `longjmp` will cause a jump to an unknown location. Again this is necessary to insure a safe approximation for data flow analyses.

Some of the *escaping branches* can be avoided by duplicating code. We may choose to do this during the editing phase if the resulting code growth is reasonable.

2.2 Editing

In the editing phase the intermediate representation is transformed to achieve the desired goal, e.g., instrumentation, optimization, or code compression.

The following chapters present a variety of concrete transformations and the analyses necessary to support them. Here we will only discuss issues of a more general nature.

In `Alto` we restrict ourselves to changing code and code-related pieces of data like jump tables. The data portion of the program is left unchanged. The reason for this is that most high level information necessary to make correct transformations within the Data Segment is either lost during compilation or is extremely hard to recover. For example, we are not able to change the orders of variables in a structure or record.

2.2.1 Scale Problems

Figure 2.1 summarizes the basic characteristics of the SPECint95 benchmark suite [65], which are used in most of the experiments in this thesis. The benchmark programs were statically compiled, and hence the numbers include library subroutines.

Benchmark	#Instructions	#Edges	#Nodes	#Subroutines
`compress`	18759	9224	5021	224
`gcc`	295096	158723	77505	2130
`go`	71721	29454	15696	605
`ijpeg`	54611	21532	11534	639
`li`	34768	17225	9138	646
`m88ksim`	46117	21940	11473	528
`perl`	90318	44997	22662	618
`vortex`	127383	58107	28465	1026

Table 2.1: Characteristics of the SPECint95 benchmarks

As can be seen from the table, some benchmarks are quite big, e.g., `gcc` has about 300,000 instructions. Its intermediate representation in `Alto` consumes almost 100MB. Hence, any algorithm applied during the editing phase must be aware of the potentially huge size of the intermediate representation. Algorithms that work well in conventional compilers (which operate on a per module or per subroutine basis) might be impractical for object code modifications because of the high time and especially the high space complexity. Clearly, any algorithm that is quadratic in the number of instructions will not be feasible. Stingy algorithms — preferably linear in both time and space — are necessary. Often we will be forced to make tradeoffs between precision and efficiency. Furthermore, memory locality of the algorithms and data structures significantly influences performance.

2.2.2 Self Modifying Code

Self modifying code and runtime generated code are a major hurdle to object code modification. Self modifying code had become somewhat out of fashion but is now gaining popularity again either directly [30, 29] or indirectly for just-in-time compilation [33].

`Alto` can cope with some forms of self modifying code, but will not work under all circumstances. Imagine a piece of code *A* in the Text Segment which is changed at runtime by another piece of code *B*. Note that this violates the read-only character of the Text Segment [4]. If we change *A* in the editing phase the assumptions made by *B* about *A* might not be valid anymore. This could cause the program to work incorrectly.

However, in the more likely case that code is generated on the fly into a dynamically allocated piece of memory there will be no problems. The code cannot be altered by `Alto` since it is not part of the Text Segment. Invocation of such code will look like an indirect control transfer. This control transfer is safely modeled using *unknownfun*, making worst case assumptions about the runtime behavior of the code.

2.3 Code Generation

After the transformations on the intermediate representation are finished, we need to generate a new version of the executable. Converting the three-address code back into machine instructions does not pose any problems. Adding the unconditional branches that have been removed during parsing is also straightforward. The difficult part lies in translating the old code addresses into new ones and dealing with changed segment sizes.

[4] Unfortunately, the programmer can use system calls to change access restrictions for segments

2.3.1 Address Translation

The problem of address translation is a consequence of the fact that, after object code modification, code addresses (in particular subroutine start addresses) will have changed. Address translation has historically been a major problem for object code modification systems. Several solutions have been proposed and implemented [75].

One approach is to avoid the problem by allowing only transformations that do not change code addresses, e.g., old code can not be deleted or new code inserted. We are allowed only to substitute code, e.g., substitute an instruction with an unconditional branch to a piece of code which executes the original instruction and after doing some extra work branches back. Clearly, this approach is only useful for instrumentation but it can be used to instrument a running program [61] in its address space, while it is running.

The second approach translates some addresses statically and others dynamically, viz. at runtime. Pc-relative branches and subroutine calls are easily handled statically; so are branches and subroutine calls to absolute addresses. Targets of these branches and procedure calls are, by definition, basic block beginnings. Therefore all the system has to do is to remember, for each basic block, the original address. After code generation the new addresses of the basic blocks are also known, and we can translate old addresses to new addresses. Other control transfer instructions, i.e., indirect control transfers, are handled by runtime address translation. Here a code snippet is added before an indirect control transfer (jump) which, with the help of an additional table, translates old addresses into new addresses at runtime. The table is an array of new addresses indexed by old addresses, and is appended to the Text Segment. If the snippet cannot find an address in the table, it leaves it unchanged. This will allow runtime generated code to work properly. This approach is not well suited for optimizing executables, but has been used for instrumentation [75, 47]

The third approach — the one used by `Alto` — does all the address translation statically. This approach makes the assumption that indirect control transfers will branch to addresses that have ultimately been loaded (verbatim) from memory. Hence all the static translator has to do is to find the memory locations containing code addresses and replace them with the corresponding new address. These memory locations — which might be in the Text Segment, Data Segment or Program Header — are identified using relocation information. This approach relies on the compiler to avoid certain coding styles that would break the scheme. For example, consider the two translations of a C switch statement using a computed jump in Figure 2.4. [5].

```
    .text
    lda  r1, table
    addq r1, r0, r1
    jmp  (r1)

table:
    br   targetA
    br   targetB
    br   targetC
```

(a) bad implementation

```
    .data
targets:
    .word targetA, targetB,
                   targetC
    .text
    lda  r1, targets
    addq r1, r0, r1
    ldq  r1, 0(r1)
    jmp  (r1)
```

(b) good implementation

Figure 2.4: Translations of a C switch statement using a computed jump

The value which is switched upon resides in register *r*0. The targets of the switch statement are the labels `targetA`, `targetB`, and `targetC` (not shown). The left

[5]The meaning of the Alpha machine instructions is explained in Appendix A

hand side solution (a) adds the value of $r0$ to the table address to obtain the target of the indirect jump. The indirect jump is followed by an unconditional branch to the final destination. The right hand side solution (b) adds the value of $r0$ to the address of a table containing the possible jump targets, then loads the target address and jumps to the final destination directly.

While there are no problems with the right hand side solution (b), the left hand side solution (a) will not work, because the target of the computed jump is the result of an arithmetic computation (rather than an indirect reference). `Alto` has no way of telling that the `table` of unconditional branches is part of a computed jump, and should therefore remain unchanged. In fact, `Alto` removes all unconditional branches from its intermediate representation and instead maintains them as edges in the control flow graph. The resulting empty nodes may be moved around and possibly merged with other non-empty nodes.

2.3.2 Segment Growing

Object code modification will usually change the size of the Text Segment. This is not a problem if the size shrinks, since we can pad it to the original length. However, if the size grows beyond the original size (because of, for example, inlining or instrumentation), the end of the new Text Segment may overlap the beginning of the Data Segment in the address space, forcing us to move the Data Segment and the BSS Segment to higher addresses. This can be achieved statically by using relocation information to identify all memory locations containing addresses inside the Data Segment or BSS Segment and updating them accordingly. Or, it can be achieved dynamically, by inserting code before all load and store instruction to update the load/store address if necessary. Luckily, under Digital Unix/Alpha, there is usually a very big gap between the end of the Text Segment

and the beginning of the Data Segment, [6] which eliminates the problem. The growth of the Text Segment might not exclusively stem from code growth: we also need extra space for new read-only constants and jump tables. For instrumentation purposes we might also want to increase the size of the Data Segment to make space for profiling counters. Again, because of the gap between the Text Segment and Data Segment under Digital Unix/Alpha, this can be easily accomplished by growing the Data Segment toward lower addresses and placing the extra data structures before the original Data Segment.

[6] The Text Segment typically start at 0x120000000, and the data segment at 0x140000000.

CHAPTER 3

ANALYSES

In this chapter we describe techniques, analyses, and data structures used by `Alto` which are useful independently of the purpose of object code modification. The main focus is on register liveness analysis, which will provide the necessary scratch registers for many transformations performed during the editing phase.

3.1 Register Liveness Analysis

Liveness 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* if this is the case. Liveness analysis of variables is a well-understood technique employed by most compilers to guide optimizations such as useless code elimination and register allocation [55]. Liveness analysis can also be performed on object code if we let registers take the place of variables. Its main purpose is to identify useless code and to provide scratch registers for the transformations performed during the editing phase.

Compared to traditional variable liveness analysis which is usually intraprocedural, the register liveness analysis for executable code presented here will be interprocedural. Interprocedural analysis on registers is simplified by the fact that there is no aliasing between registers and the number of registers for any given processor is bounded by a constant. What makes it difficult are control flow anomalies (cf. Section 2.1.4) and scale issues (cf. Section 2.2.1).

Related Work: Work most closely related to our own has been done by Srivastava and Wall on the `OM` optimizer [67] and by Goodwin on the `Spike` optimizer [36]. We improve on their liveness analysis in three ways. Firstly, we have changed the underlying flow equations resulting in three sets of almost identical equations, which simplifies implementation and reasoning about correctness. Secondly, we accelerate the fixpoint iteration by exploiting a novel insight about the interdependence of the various pieces of data flow information. This idea is also applicable to liveness analysis of variables. Thirdly, we show how to reduce the space requirement of the analysis by recomputation and exploitation of the new data flow equations.

Furthermore, we explore ways to improve the accuracy of liveness analysis. For a known technique involving callee-save registers we point out a possible generalization.

3.1.1 Interprocedural Data Flow Analyses

Intraprocedural data flow analyses consider all possible paths in the control flow graph of a subroutine to give an estimate of what data flow facts hold at a given node. Conditionals are not interpreted, i.e. we assume that both sides of the branch can always be taken. As a result, we may include paths that will never be executed in reality and the estimate will be somewhat conservative.

For interprocedural data flow analyses we can simply adopt the intraprocedural approach and regard the interprocedural control flow graph as one big ordinary control flow graph, treating *call* and *return* edges as regular edges and ignoring *link* edges. Analyses performed in this fashion are called *context insensitive* interprocedural analyses. Such analyses are simple and fast but often yield rather conservative estimates since many paths in the interprocedural control flow graph do not reflect real program executions. An example is shown in Figure 3.1 where two call sites call the same subroutine f. Consider the path $C1 \rightarrow IN \rightarrow EX \rightarrow R2$. This path returns to the wrong call site and hence

does not occur in any execution. But since variable $s2$ is used in $R2$ and not defined along the path we conclude that $s2$ is live at $C1$, while in fact $s2$ is dead, as it is defined in $R1$.

Paths which do not return to the wrong call site are called *realizable paths*, e.g., $C1 \rightarrow IN \rightarrow EX \rightarrow R1$ or $C2 \rightarrow IN \rightarrow EX \rightarrow R2$. See [46] for a more rigorous definition.

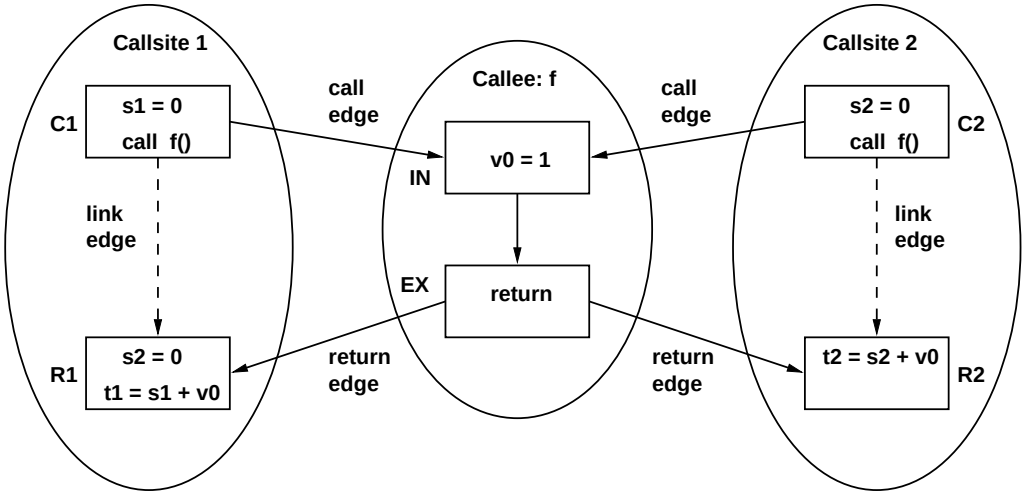


Figure 3.1: Unrealizable path in context insensitive analyses

A context sensitive interprocedural data flow analysis considers only realizable paths in the interprocedural control flow graph [52].

3.1.2 Interprocedural Register Liveness Analysis

In this section we discuss two flavors of interprocedural liveness analysis: Context sensitive and context insensitive. Tuning possibilities are described and performance numbers presented.

3.1.2.1 Context Insensitive Analysis

As described in the previous section, the context insensitive liveness analysis uses the standard intraprocedural analysis [55] and applies it to a program's interprocedural control flow graph treating *call* and *return* edges as ordinary edges, and ignoring *link* edges.

The analysis iteratively computes the fixpoint of the equations below

$$LiveIn[n] = use[n] \cup (LiveOut[n] - def[n]) \quad n \in Nodes$$

$$LiveOut[n] = \bigcup s \in Succ[n] : LiveIn[s] \quad n \in Nodes$$

subject to the initial values

$$LiveOut[n] := \emptyset \quad n \in Nodes$$

$$LiveIn[n] := \emptyset \quad n \in Nodes$$

$\mathcal{R}$ denotes the set of all registers. For each node *n*, $LiveIn[n]$ ($LiveOut[n]$) contains the registers live at the beginning (end) of the node, $def[n]$ contains the registers which are defined in *n*, $use[n]$ contains the registers which are used before they are defined in *n*.

3.1.2.2 Context Sensitive Analysis

For context sensitive liveness analysis we must restrict ourselves to realizable paths through the interprocedural control flow graph. This is achieved by considering intraprocedural paths only and modeling subroutine calls using summary information for the called subroutine [52]. Conceptually, all *call* and *return* edges are removed from the interprocedural control flow graph. Data flow through *link* edges is subject to modifications described by the summary information for the called subroutine.

Two pieces of information are necessary to summarize the effects of each subroutine f on liveness:

- $MayUse[f]$. The set of registers that may be used by f. A register *r* may be used by f if there is a realizable path from $InitNode[f]$ to a use of *r* without an intervening definition of *r*. $MayUse[f]$ hence describes the set of registers which are live at the

beginning of $InitNode[f]$ independent of the calling context and hence are live at the end of any *call* node n_c calling f. Typically these are the registers which are used to pass arguments to subroutine f.

- $ByPass[f]$. The set of registers which if live at n_r are live at n_c for any *call* node n_c calling f. Typically these are the register which are not used at all by f.

We also define

- $MustDef[f]$. The set of registers which are defined (written to) on all paths from $InitNode[f]$ to $ExitNode[f]$.
- $MustDead[f]$. The set of registers which are defined on all paths from $InitNode[f]$ to $ExitNode[f]$ and are not used before they are defined. Clearly, $MustDead[f] = MustDef[f] - MayUse[f]$

Once *ByPass* and *MayUse* information has been computed for each subroutine, liveness information is computed as follows:

Phase *Live* : *Computation of LiveIn and LiveOut*:

iteratively compute the fixpoint of the data flow equations listed below

$$
\begin{array}{lll}
LiveIn[n] & = use[n] \cup & n \in Nodes \\
 & \quad (LiveOut[n] - def[n]) & \\
LiveOut[n] & = \bigcup s \in Succ[n] : & n \in Nodes \ \wedge \ Type[n] \notin \{call\} \\
 & \quad LiveIn[s] & \\
 & = MayUse[f] \cup & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & \quad (ByPass[f] \cap LiveIn[n']) & n' = ReturnNode[n] \ \wedge \ f = Callee[n]
\end{array}
$$

subject to the initial values

$$
\begin{array}{lll}
\mathit{LiveOut}[n] & := \emptyset & n \in \mathit{Nodes} \\
\mathit{LiveIn}[n] & := \emptyset & n \in \mathit{Nodes} \\
\mathit{MayUse}[f] & := \text{as computed by Phase } \mathit{MayUse} & f \in \mathit{Functions} \\
\mathit{ByPass}[f] & := \text{as computed in Phase } \mathit{ByPass} & f \in \mathit{Functions}
\end{array}
$$

The important aspect is the computation for the *call* nodes. A register is live at the end of a *call* node if the register is used by the callee (*MayUse* case) or it is live at the corresponding *return* node and not defined — on at least one realizable path — inside of the callee (*ByPass* case). This gives us some choice in the selection/computation of the *ByPass* sets. If a register is in $\mathit{MayUse}[f]$ we can include it in $\mathit{ByPass}[f]$ even if the register is never live at any of the corresponding *return* nodes. Srivastava *et al.* [67] choose $\mathit{ByPass}[f]$ to be $\overline{\mathit{MustDead}}[f]$. The problem with this approach is that it introduces a mutual dependency between *ByPass* information and *MayUse* information which complicates the flow equations. Goodwin [36] chooses $\mathit{ByPass}[f]$ to be $\overline{\mathit{MustDef}}[f]$ which does not have this problem and is therefore preferable. In fact, any set which lies between $\overline{\mathit{MustDef}}[f]$ and $\overline{\mathit{MustDef}}[f] \cup \mathit{MayUse}[f]$ is a valid candidate for $\mathit{ByPass}[f]$. Our choice for $\mathit{ByPass}[f]$ is a superset of Goodwin's [1] and will result in more uniform data flow equations. Below we show how the *ByPass* and *MayUse* sets are computed.

Phase *MayUse* : *Computation of* $\mathit{MayUse}[f]$:

iteratively compute the fixpoint of the data flow equations listed below

[1] it is difficult to give more intuitive description for this choice other than the fixpoint equations

$$
\begin{array}{lll}
MayUseIn[n] & = use[n] \cup & n \in Nodes \\
 & \quad (MayUseOut[n] - def[n]) & \\
MayUseOut[n] & = \bigcup s \in Succ[n] : & n \in Nodes \ \wedge \ Type[n] \notin \{call, exit\} \\
 & \quad MayUseIn[s] & \\
 & = MayUse[f] \cup & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & \quad (ByPass[f] \cap MayUseIn[n']) & n' = ReturnNode[n] \ \wedge \ f = Callee[n] \\
MayUse[f] & = MayUseIn[InitNode[f]] & f \in Functions
\end{array}
$$

subject to the initial values

$$
\begin{array}{lll}
MayUseOut[n] & := \emptyset & n \in Nodes \\
MayUseIn[n] & := \emptyset & n \in Nodes \\
MayUse[f] & := \emptyset & f \in Functions \\
ByPass[f] & := \text{as computed in Phase } ByPass & f \in Functions
\end{array}
$$

Phase *ByPass* : *Computation of ByPass*$[f]$:

iteratively compute the fixpoint of the data flow equations listed below

$$
\begin{array}{lll}
ByPassIn[n] & = use[n] \cup & n \in Nodes \\
 & \quad (ByPassOut[n] - def[n]) & \\
ByPassOut[n] & = \bigcup s \in Succ[n] : & n \in Nodes \ \wedge \ Type[n] \notin \{call, exit\} \\
 & \quad ByPassIn[s] & \\
 & = (ByPass[Callee[n]] \cap & n \in Nodes \ \wedge \ Type[n] = call \ \wedge \\
 & \quad ByPassIn[n']) & n' = ReturnNode[n] \\
ByPass[f] & = ByPassIn[InitNode[f]] & f \in Functions
\end{array}
$$

Subject to the initial values

$$
\begin{array}{lll}
ByPassOut[n] & := \emptyset & n \in Nodes \wedge \ Type[n] \neq exit \\
 & := \mathcal{R} & n \in Nodes \wedge \ Type[n] = exit \\
ByPassIn[n] & := \emptyset & n \in Nodes \\
ByPass[f] & := \emptyset & f \in Functions
\end{array}
$$

Contrary to the intraprocedural liveness analysis or the context insensitive analysis the choice of the starting values is crucial, e.g., initializing $ByPassOut[f]$ of non-*exit* nodes to $\mathcal{R}$ as in [36] yields overly conservative results [37]. Differing from Goodwin's approach we have modified the equation for $ByPassIn[n]$ by adding (unioning) $use[n]$ to the right hand side. This makes our *ByPass* sets strictly bigger than his but since $use[n] \subseteq MayUseIn[n]$ holds, $ByPassIn[InitNode[f]]$ will still lie between $\overline{MustDef}[f]$ and $\overline{MustDef}[f] \cup MayUse[f]$. The major virtue of this change is that it makes the equations of the three phases sufficiently similar that they can be unified into just one simple and compact set of equations (cf. Figure 3.2). The code implementing the analysis, which uses the unified equations by means of a subroutine call is also correspondingly simpler and smaller. The bigger sets do not affect the performance if they are realized as bit vectors.

Unified Dataflow Equations:

$$
\begin{array}{lll}
DataIn[n] & = use[n] \cup (DataOut[n] - def[n]) & n \in Nodes \\
DataOut[n] & = \bigcup s \in Succ[n] : DataIn[s] & n \in Nodes \ \wedge \ Type[n] \notin \text{NoTypes} \\
 & = MayUse[f] \cup (ByPass[f] \cap & n \in Nodes \ \wedge \ Type[n] = call \\
 & \quad DataIn[ReturnNode[n]]) & \\
Summary[f] & = DataIn[InitNode[f]] & f \in Functions
\end{array}
$$

Unified Initial Values:

$$
\begin{array}{lll}
DataOut[n] & := \emptyset & n \in Nodes \wedge \ Type[n] \neq exit \\
 & := \text{ExitData} & n \in Nodes \wedge \ Type[n] = exit \\
DataIn[n] & := \emptyset & n \in Nodes \\
Summary[f] & := \emptyset & f \in Functions
\end{array}
$$

Phase Adaptations:

	DataIn	DataOut	NoTypes	Summary	ExitData
Phase *ByPass*	*ByPassIn*	*ByPassOut*	$\{call, exit\}$	*ByPass*	$\mathcal{R}$
Phase *MayUse*	*MayUseIn*	*MayUseOut*	$\{call, exit\}$	*MayUse*	$\emptyset$
Phase *Live*	*LiveIn*	*LiveOut*	$\{call\}$	—	$\emptyset$

Figure 3.2: Unified fixpoint computation

3.1.2.3 Tuning the Context Sensitive Analysis

Even though our presentation of the data flow equations as phases suggest a certain ordering of execution, we can compute all fix points simultaneously because all equations are monotone. However, if executed sequentially (in the order *ByPass*, *MayUse*, *Live*) the space used to hold $ByPassOut[n]$ and $ByPassIn[n]$ can be re-used to hold $MayUseOut[n]$ and $MayUseIn[n]$ which in turn can be reused to hold $LiveIn[n]$ and $LiveOut[n]$. For the SPEC95 benchmark `gcc` the total amount of memory needed to hold each of the *ByPass*, *MayUse*, and *Live* fields is about 600 kB. Re-using space will reduce memory require-

ments and also improve memory locality. When comparing Phase *MayUse* with Phase *Live* it becomes evident that the fixpoint for $LiveOut[n]$ (resp. $LiveIn[n]$) must be a superset of the fixpoint for $MayUseOut[n]$ (resp. $MayUseIn[n]$). Hence, it is safe to initialize $LiveIn[n] := MayUseIn[n]$ and $LiveOut[n] := MayUseOut[n]$ thereby accelerating Phase *Live* by not having to start the fixpoint iteration from scratch.

Next we describe how to improve Phase *Live* more drastically exploiting the following observation. We focus on Out-sets here; In-sets are analogous. For a register r at node n of subroutine f, we have

$$r \in LiveOut[n] \Rightarrow r \in MayUseOut[n] \ \vee \ r \in ByPassOut[n]$$

Conversely,

$$r \in MayUseOut[n] \Rightarrow r \in LiveOut[n]$$

But $r \in ByPassOut[n] \not\Rightarrow r \in LiveOut[n]$. The latter does not hold because our initial values for *ByPassOut* of the *exit* nodes were pessimistic; we essentially assumed that all registers could be live. During Phase *Live* it might turn out that not all registers are live at some *exit* nodes. The correct condition is therefore

$$r \in ByPassOut[n] \ \wedge \ r \in LiveOut[ExitNode[f]] \Rightarrow r \in LiveOut[n] \qquad (3.1)$$

This suggests the following alternative approach for Phase *Live*, which has the benefit of iterating only over the *exit* nodes of the intraprocedural control flow graph.

```
(1) FOREACH n∈Nodes DO
(2)   LiveOut[n]:= MayUseOut[n]
(3)   LiveIn[n] := MayUseIn[n]
(4) REPEAT
(5)   changed := false
```

```
 (6)   FOREACH f∈Function DO
 (7)     new_out := ⋃ s∈Succ[ExitNode[f]] : LiveIn[s]
 (8)     IF new_out ≠ LiveOut[ExitNode[f]] THEN
 (9)       changed := true
(10)       LiveOut[exit[f]] := new_out
(11)       FOREACH n∈Nodes[f] DO
(12)         LiveOut[n]:= MayUseOut[n]∪(ByPass[n]∩new_out)
(13)         LiveIn[n] := MayUseIn[n] ∪(ByPass[n]∩new_out)
(14) UNTIL ¬changed
```

We begin by setting the start values for the fixpoint iterations using the improvement mentioned above (Lines 1 through 3). Then, we recompute the liveness information at the *exit* nodes for all functions until there is no change (Lines 4-14). If the liveness information at an *exit* node has changed we propagate this change according to (3.1) to all nodes of this subroutine (Lines 11 through 13). Note that it suffices to propagate this information to *return* nodes only.

LiveOut and *MayUseOut* (resp. *LiveIn* and *MayUseIn*) need not be kept in separate locations; they can be merged into one, i.e., all occurrences of *LiveOut* (resp. *LiveIn*) can be replaced by *MayUseOut* (resp. *MayUseIn*) which will then contain the liveness information upon completion of the fixpoint iteration. This also renders the first three lines of the algorithm unnecessary.

Since Phase *Live* is usually the costliest of the three, this improvement cuts down execution time by 25%. (cf. Section 3.1.2.5 for experiments results). The drawback is that space usage almost doubles because both *ByPass* and *MayUse* information have to be kept around for each node (assuming *Live* information has been merged with *MayUse*

information).

The enhancement is also applicable to ordinary interprocedural liveness analyses of variables.

3.1.2.4 Control Flow Anomalies

Control flow anomalies as described in Section 2.1.4 are automatically handled by associating worst case assumptions with *unknownnode* and *unknownfun* as shown below and relying on the presence of compensation edges.

$$
\begin{array}{llll}
ByPassOut[unknownnode] & := \mathcal{R} & LiveOut[unknownnode] & := \mathcal{R} \\
ByPassIn[unknownnode] & := \mathcal{R} & LiveIn[unknownnode] & := \mathcal{R} \\
MayUseOut[unknownnode] & := \mathcal{R} & MayUse[unknownfun] & := \mathcal{R} \\
MayUseIn[unknownnode] & := \mathcal{R} & ByPass[unknownfun] & := \mathcal{R}
\end{array}
$$

3.1.2.5 Implementation and Performance of the Liveness Analyses

We have implemented the context sensitive and context insensitive liveness analysis algorithms within `Alto`. Besides the speed of the analysis, space consumption was of primary concern to us. We found that it is usually better to recompute a data item than to store it. Thus, `Alto` only stores the various *Out*-sets associated with a node. The *In*-sets are computed by traversing the instructions of a basic block backwards. [2] The *def* and *use* sets are not needed at all.

The relatively small number of instructions in a typical node make this approach viable. We also do not maintain a worklist of those nodes that need to be reconsidered during the fixed point iteration because this would incur the cost of at least one more pointer per node. Instead, we mark those nodes which need recomputation and iterate

[2] Alternatively, we could keep the *IN*-sets and recompute the *Out*-sets from the successor nodes. However, when an optimization needs to determine which registers are live at a point within a node, it is more convenient to have the *Out*-sets readily available.

over all nodes, processing marked nodes until no marked ones are left.

The total space requirement for the context insensitive liveness analysis is 64 bits per node to hold the *LiveOut* information. (1 bit for each of the 64 registers of the Alpha CPU). For the context sensitive analysis running the three phases sequentially we need an additional 128 bits per function to hold the *ByPass* and *MayUse* summary information simultaneously. For the improved version of the context sensitive analysis described in the previous section, we need an additional 64 bits per node because we need to access *MayUseOut* and *ByPassOut* simultaneously.

Our experiments are based on the SPECint95 benchmark suite. Figure 2.1 summarizes their basic characteristics.

Figure 3.1 shows our experimental results for the liveness analyses. The measurements were obtained on our reference machine (cf. Section 4.1). Besides time and space usage, we also measured the precision. For the improved context sensitive analysis, the space and time requirements are given in square brackets (the precision is not affected). The precision is computed as the average number of dead integer registers after all instructions, i.e., the number of integer registers not live averaged over all program points.

The last column contains the difference in precision between the context sensitive and insensitive analysis.

Benchmark	Context Insensitive			Context Sensitive [improved]			Δ
	Sp. (kB)	Ti. (sec)	Prec.	Space (kB)	Time (sec)	Prec.	Prec.
`compress`	39	0.05	4.9	42 [81]	0.15 [0.10]	6.6	1.7
`gcc`	605	1.30	4.2	638 [1244]	3.75 [3.00]	6.6	2.5
`go`	122	0.20	5.9	132 [254]	0.55 [0.40]	11.5	5.7
`ijpeg`	90	0.15	4.6	100 [190]	0.40 [0.30]	5.4	0.8
`li`	71	0.10	3.7	81 [152]	0.30 [0.20]	5.4	1.8
`m88ksim`	89	0.15	4.9	97 [187]	0.35 [0.25]	7.1	2.3
`perl`	177	0.30	4.3	186 [363]	0.85 [0.65]	6.2	1.9
`vortex`	222	0.45	5.3	238 [460]	1.30 [1.00]	8.6	3.3

Table 3.1: Performance of liveness analysis

The context sensitive analysis typically finds two additional dead integer registers per node over the insensitive analysis and takes roughly three times as long to compute. Our improvement to the context sensitive analysis speeds the computation up by approximately 25% at the cost of a roughly twice the memory usage.

The number of available dead register suggest that usually there are plenty of scratch registers available for program transformation such as the insertion of instrumentation code.

3.1.3 Improving the Precision of Register Liveness Analysis

This section explores how the precision of liveness analysis can be improved. An obvious source for improvement is our overly pessimistic treatment of *unknownnode* and *unknownfun*. This will be exploited in 3.1.3.2. Section 3.1.3.1 shows how some registers

which the analysis correctly identified as live can nevertheless be regarded as dead in some contexts.

3.1.3.1 Callee Save Registers

As described by Goodwin in [36], information about callee save registers can be exploited to reduce the number of live registers. Let $Saved[f]$ denoted the registers which are saved and restored by f and which are otherwise not used before defined in f.[3] $Saved[f]$ will be a subset of $MayUse[f]$ because the saving of a register at function entry will be regarded as a use of that register by the liveness analysis. However, this use is only relevant if the register is live at the return node of a given call site.

Hence we can remove $Saved[f]$ from $MayUse[f]$ and instead add it to $ByPass[f]$ without affecting safety. The following slight modification of the equations updating the summary information in Phase *ByPass* and *MayUse* achieves the desired effect.

$$ByPass[f] = ByPassIn[InitNode[f]] \cup Saved[f] \qquad f \in Functions$$

$$MayUse[f] = MayUseIn[InitNode[f]] - Saved[f] \qquad f \in Functions$$

In order to get a better insight into how this optimization opportunity arises and how it may be generalized we consider the following (hypothetical) code for complex addition and two of its call sites. [4].

[3] As described in the next section we cannot rely on the calling conventions when determining $Saved[f]$ but need to inspect f

[4] The meaning of the Alpha machine instructions is explained in Appendix A

```
ComplexAdd:
    addq  r10, r12, r0
    addq  r11, r13, r1
    ret   ra

Callsite2:
    ...
    bsr   ra, ComplexAdd
    mulq  r0, r0, r0
    mulq  r1, r1, r1
    addq  r0, r1, r0
    ...
    ret   ra
```

```
Callsite1:
    ...
    ldq  r10, 0(r20)
    ldq  r11, 8(r20)
    ldq  r12, 0(r21)
    ldq  r13, 8(r21)
    bsr  ra, ComplexAdd
    move r0, r10
    bsr  ra, PrintNumber
    ...
    ret ra
```

Figure 3.3: Code example: addition of complex numbers

For `ComplexAdd` the real and imaginary part of the first summand is passed in registers *r*10 and *r*11, the real and imaginary part of the second summand in registers *r*12 and *r*13, and the result is returned in *r*0 and *r*1. `Callsite1` just prints out the real part of the result, while `Callsite2` computes its squared norm.

Clearly, registers *r*10 through *r*13 will be live at both call sites just before the call to `ComplexAdd`. But `Callsite1` only uses the real part of the result hence the result computed by the second add in `ComplexAdd` is useless. A lazy programming language would neither execute this add instruction nor the instructions computing the values of registers *r*11 and *r*13. Unfortunately, we cannot eliminate the `addq` instruction since `Callsite2` uses both results *r*0 and *r*1. But since *r*1 is dead in `Callsite1` we

can consider registers $r11$ and $r13$ to be dead as well and subsequently eliminate the corresponding load instructions. Registers $r11$ and $r13$ will then have arbitrary values and the add instruction produces an arbitrary result which is ignored [5]

In this light the callee save register s can be regarded as an additional argument a_s and result r_s of the function f. (s, a_s, and r_s will of course denote the same register.) a_s will be moved to a new location and then from there to r_s. If r_s is not live at a given return node the move operations are useless. But as above we cannot delete them. All we can do is mark a_s as dead at the corresponding call node and this is exactly what is achieved by moving s from $MayUse[f]$ to $ByPass[f]$.

3.1.3.2 Calling Conventions

Suppose function f does not use or define register r and does not call any other function. Our liveness analysis will determine that $r \in ByPass[f]$. Now assume that f and any function calling f obey some sort of calling convention which state that register r is not preserved across procedure calls and does not carry a result. This implies that r will not be live at any return node of a call site of f and it is therefore safe to remove r from any $ByPass[f]$. In fact, it is irrelevant whether r is in $ByPass[f]$ or not. The smaller *ByPass* set is nevertheless desirable, because *unknownnode* or *unknownfun* may introduce unwanted liveness information into the analysis which would be partially eliminated by the smaller set. Unfortunately, we have no control over the enforcement of calling conventions in general, except for system calls. In fact, compilers often violate calling convention when they perform interprocedural register allocation or when library functions are invoked that implement missing hardware features such as a divide instruction. It seems reasonable, however, to assume that calls to shared libraries and calls through function pointers respect the calling convention.

[5] If `addq` could cause a side effect such as an overflow this approach is of course not valid.

In our current version of the liveness analysis those calls are modeled by a call to *unknownfun*. An enhancement would be to model this as a call to a different function *sysfun* (or a special node *sysnode* for a context insensitive analysis). [6]

Let *sysuse* denote the set of registers potentially used according to the calling conventions and *syssave* the set of registers preserved across function calls. The liveness analysis will be augmented with the following assignments.

$$
\begin{array}{llll}
\mathit{ByPassOut}[\mathit{sysnode}] & := \mathit{syssave} & \mathit{LiveOut}[\mathit{sysnode}] & := \mathit{sysuse} \cup \mathit{syssave} \\
\mathit{ByPassIn}[\mathit{sysnode}] & := \mathit{syssave} & \mathit{LiveIn}[\mathit{sysnode}] & := \mathit{sysuse} \cup \mathit{syssave} \\
\mathit{MayUseOut}[\mathit{sysnode}] & := \mathit{sysuse} & \mathit{MayUse}[\mathit{sysfun}] & := \mathit{sysuse} \\
\mathit{MayUseIn}[\mathit{sysnode}] & := \mathit{sysuse} & \mathit{ByPass}[\mathit{sysfun}] & := \mathit{syssave}
\end{array}
$$

3.1.3.3 Performance

We have added the enhancements described in the previous sections to the context sensitive analysis and measured the resulting gain in precision. Figure 3.4 shows the average number of dead integer registers after all instructions without any enhancement, with one of the enhancements, and with both enhancements.

Our experiments show that incorporating both enhancements increases the number of dead registers by as much as 10. Most of the improvement is due to the calling convention enhancement. The reason for this lies with a mechanism related to the C standard library function `atexit()`. This function lets the programmer register other function which are called upon termination of the program, i.e., typically after return from function `main()`, using function pointers. Without the calling convention enhancement the function pointer invocation will cause all register to be live at the end of `main()` and this will propagate backwards into most other subroutines.

[6] If the calling conventions for system calls differ from those for regular functions, as it is the case for Digital Unix/Alpha, we introduce one function/node for each calling convention

Benchmark	None	Save	Call. Conv.	Both	Both - None
`compress`	6.6	6.8	16.5	17.1	10.5
`gcc`	6.6	7.2	11.2	12.1	5.4
`go`	11.5	11.7	16.8	17.1	5.5
`ijpeg`	5.4	5.5	15.5	15.7	10.3
`li`	5.4	5.8	15.3	15.7	10.3
`m88ksim`	7.1	7.4	16.5	17.4	10.2
`perl`	6.2	6.4	15.4	15.9	9.7
`vortex`	8.6	9.2	15.7	17.0	8.4

Figure 3.4: Impact of enhancements to liveness analysis

3.2 Register Use-Def Chains

Register use-def chains provide, for each use of a register, a pointer to its definition. A use of a register occurs when an instruction uses a register as its operand, i.e., reads that register. A definition of a register is an instruction that defines (writes) a register. The use-def chains are a directed graph whose nodes are instructions and whose edges are use-def pointers. In order to preserve space we only allow for one pointer per use. If there are several definitions of a register reaching a use as depicted on the left hand side in Figure 3.5 for register $r0$ we introduce a pseudo instruction at an appropriate place which also defines that register, thereby shadowing the other definitions. The pseudo instruction does not use any register. The resulting data structure will be cycle free because

- All registers must be defined before they are used (enforced by inserting pseudo instructions at all *init* nodes).

- All code is reachable (enforced by removing unreachable code).
- No use has more than one definition (enforced by pseudo instructions).

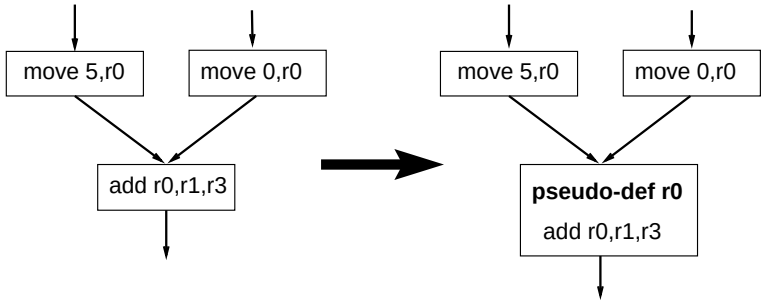


Figure 3.5: Introduction of pseudo definitions

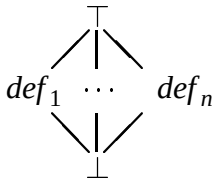
This is analogous to ϕ functions used with the static single assignment (SSA) form [25]. Use-def chains simplify the implementation of optimizations such as common subexpression elimination and analyses such as alias analysis (cf. Section 3.3).

3.2.1 Algorithm

The difficult aspect of use-def chains is to determine where to insert pseudo instructions. The algorithm proposed in [25] uses sophisticated data structures such as dominator frontiers which are efficient but quite memory intensive. Our implementation takes a different route and uses a somewhat less time but more memory efficient algorithm based on and idea by Shapiro [63, 51].

Our approach treats each register *r* separately. An intraprocedural forward data flow analysis propagates definitions of the register to its uses. If several definitions reach a use we have not yet inserted enough pseudo instructions defining that register. We insert pseudo instructions at appropriate confluence points and then restart the algorithm. Originally there is only one pseudo instruction for *r* at the *init* node of each subroutine.

More formally, we use a forward data flow analysis on the lattice depicted below. The meet operator is $\wedge$.



where def_i denotes an instruction or pseudo instruction defining r.

Phase 1: For each subroutine f iteratively compute the fixpoint of the data flow equations listed below.

$$Out[n] = \text{IF } def[n] = \bot \text{ THEN } In[n] \text{ ELSE } def[n] \text{ ENDIF} \quad n \in Nodes[f]$$

$$In[n] = \bigwedge_{p \in Pred[n]} Out[p] \quad n \in Nodes[f]$$

Subject to the initial values

$$In[n] := \bot \quad n \in Nodes[f]$$

$$Out[n] := \bot \quad n \in Nodes[f]$$

$$def[n] := \text{last definition of register } r \text{ in } n \text{ or } \bot \text{ otherwise} \quad n \in Nodes[f]$$

After the fixpoint computation $Out[n] \neq \bot$ and $In[n] \neq \bot$ holds for all $n \in Nodes[f] \setminus InitNode[f]$.

Phase 2: Determine confluence points and insert pseudo instructions.

Insert pseudo definitions in those nodes n that do not have one already and that have at least one predecessor that does not propagate $\top$ into n.

```
(1)  FOREACH n∈Nodes[f] DO
(2)     IF In[n]≠ ⊤ THEN CONTINUE ENDIF
(3)     IF n has pseudo instruction THEN CONTINUE ENDIF
(4)     FOREACH p∈Pred[n] DO
(5)        IF Out[p]≠ ⊤ THEN
```

```
 (6)           {Prepend pseudo instruction to n}
 (7)           BREAK;
 (8)        ENDIF
 (9)     ENDFOR
(10)  ENDFOR
```

Phase 3: Repeat steps 1 and 2 until no more pseudo instructions are added.
Eventually all nodes *n* with $In[n] = \top$ should contain a pseudo instruction at the beginning. This means that from a data flow point of view $\top$ is prevented from reaching any ”use” of the register *r* and all the pointers will point to a valid definition.
Phase 4: Propagate $In[n]$ to all the uses within *n*.

```
 (1)  FOREACH n∈Nodes[f] DO
 (2)     def := In[n]
 (3)     ITERATE i FORWARD THROUGH Instructions[n]
 (4)        {if i uses r set use-def-pointers to def}
 (5)        IF {i defines r} THEN
 (6)           def := i
 (7)        ENDIF
 (8)     ENDITERATE
 (9)  ENDFOR
```

3.2.2 Performance

The space requirements for the analysis is very moderate and consists of one word (4 byte) per basic block *n* to hold $In[n]$ and another word to hold $def[n]$. $Out[n]$ is dynamically computed from $In[n]$ and $def[n]$ and not explicitly stored. We also maintain a bit

vector (64 bits) per node that describes for which of the 64 registers pseudo instructions were prepended to the node. This keeps us from having to actually insert the pseudo instructions. The total space requirements for each SPECint95 benchmark is shown in column 2 in Table 3.2. It also give the execution time on our reference machine (cf. Section 4.1) in column 3. Column 4 contains the number of pseudo definitions generated and column 5 the total number of instructions for comparison.

Benchmark	Space (kB)	Time (sec)	#pseudo Defs.	# Instructions
compress	39	0.15	2818	18759
gcc	605	3.75	47754	295096
go	122	0.55	8357	71721
ijpeg	90	0.40	7947	54611
li	71	0.30	6513	34768
m88ksim	89	0.35	6179	46117
perl	177	0.85	15055	90318
vortex	222	1.30	19529	127383

Table 3.2: Performance of use-def chains

3.3 Register Alias Analysis

The problem of alias analysis or memory disambiguation at the machine code level is to determine the relationship of two memory regions, i.e., whether they are identical, disjoint, or intersecting. As a possible result we also allow the conservative estimate that nothing is known about the relationship of the two regions. The regions are typically identified with an instruction, e.g., the store instruction `stq r3,8(r11)` describes the region pointed to by $r11$ with an 8 byte offset. The region is one quadword (or 8 bytes)

wide.

Memory disambiguation is one of the weak points of object code modification because lots of the high level information available in an ordinary compiler, such as *types*, that would be greatly beneficial is unavailable. Various alias analyses have been implemented and tried within `Alto`. An early version is described in [28]. Here we describe only the current implementation.

3.3.1 Alias Analysis by Inspection

The current version of alias analysis is essentially an analysis by inspection, i.e., we try to derive a symbolic description for each of the memory regions and then compare these descriptions. A few short cuts from this general approach are taken when possible.

Stack pointer vs. other register: If one memory region is a stack location and the containing function does not make use of references into the stack, then this region can never intersect with a non-stack region.

Stack pointer vs. known address: If one memory region is a stack location and the other region lies inside the Text, Data, or BSS Segment, then the regions must be disjoint.

General case: In the general case we employ the use-def chains form Section 3.2 for our analysis. We describe our algorithm by the example given in Figure 3.6. We are interested in the relationship of the memory region accessed by the last two instructions (labeled 44 and 45).

The algorithm tries to symbolically express the start address of a region by tracing the use-def chains, which are depicted as arrows (some use-def relationships are omitted to avoid clutter).

We start with $8 + r11_{43}$, the region accessed by instruction 44. The subscript 43 for $r11$ means that it was defined at point 43. Since the instruction at point 43 has outgoing edges we can symbolically expand this to $8 + r5_{41} + r1_{16}$. 41 is merely a pseudo defi-

nition and hence we are not able to expand $r5_{41}$ any further. But $r1_{16}$ can be expanded and we obtain $136 + r5_{41} + r0_{15}$. No further expansion are possible. Analogously, processing the region accessed by instruction 12 yields $16 + r5_{41} + r0_{15}$. The two regions only differ in their constant term and so we conclude that the regions must be disjoint. If the regions had differed in any other term but the constant term the relationship of the memory regions would have been conservatively estimated as unknown.

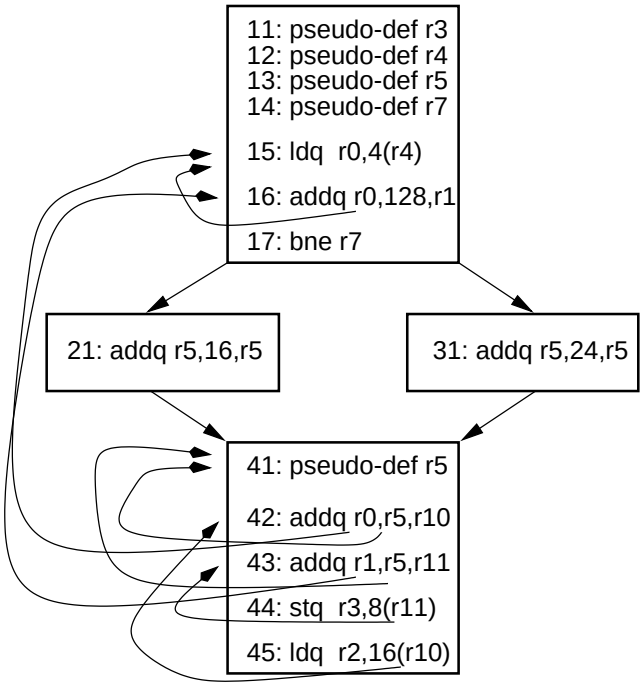


Figure 3.6: Example for alias analysis

CHAPTER 4

OPTIMIZATIONS

This chapter describes the implementation and experimental results of an optimizer based on `Alto`. The overall structure of the optimizer consists of five phases and is depicted in Figure 4.1.

Base Optimizations. After reading in the executable and transforming it into an intermediate form a series of base optimization is performed. These include most of the classical compiler optimizations such as constant folding, unreachable code elimination, copy propagation, etc. These optimization are iterated until either a fixpoint is reached or a maximum iteration count is exceeded. A second round of base optimizations is performed just before code positioning.

One-time Optimizations. This phase performs optimizations that should only be done once. There are three reasons for performing certain optimizations only once: (1) The optimization may require costly analyses (e.g., common case specialization); (2) Repetition of the optimization might have undesirable side effects (e.g., stack explosion for repeated inlining with stack merging); (3) Repeating the optimization will not give any additional benefit.

Code Positioning. After all optimizations have been executed, the interprocedural control flow graph is arranged into a linear sequence of nodes. Unconditional branches which were eliminated when the intermediate form was created are reintroduced

where necessary. Code positioning is intended to improve instruction cache hit rates and reduce the (dynamic) number of (taken) branches.

Scheduling. Scheduling reorders the instructions inside a node in order to improve the performance of a pipelined CPU. Our scheduler is a slight extensions of a regular list scheduler and allows instructions to move into other nodes if this preservers correctness of the program.

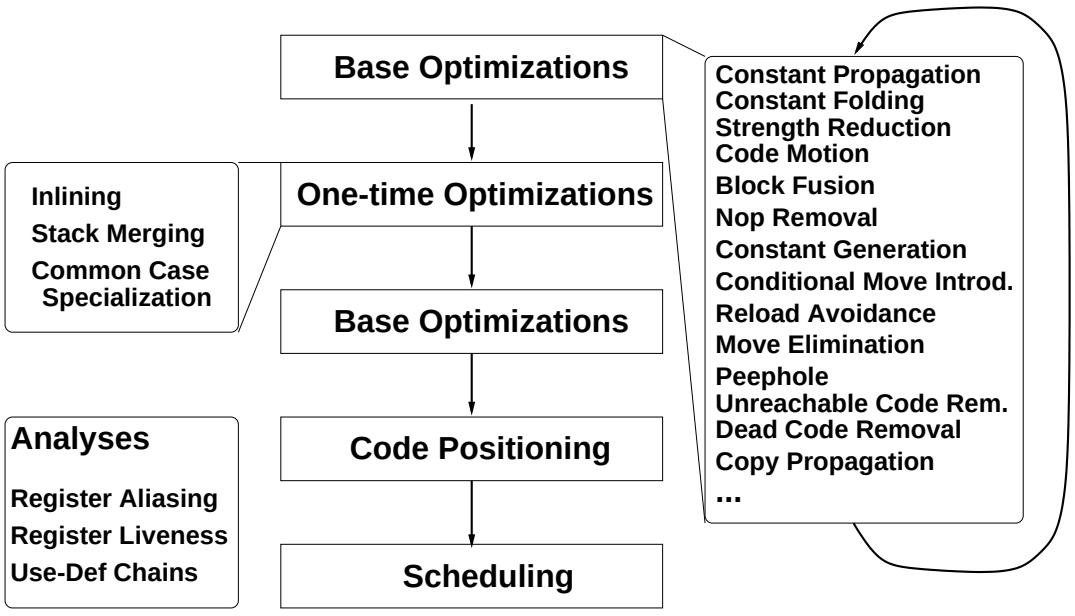


Figure 4.1: Phases of the optimizer based on `Alto`

The optimizations are supported by the analyses described in the previous chapter.

4.1 Experimental Setup

The following sections describe the most relevant optimizations performed by the optimizer and evaluates their effectiveness on the SPECint95 benchmark suite [65]. Unless

otherwise noted, the benchmarks were compiled with the DEC C compiler V5.2-036 invoked as `cc -O4`, with additional linker options (`-d -z -r -non_shared`) to retain relocation information and produce statically linked executables.[1] By default the optimizer uses execution frequency profiles obtained with the training input for these benchmarks. The execution times reported were generated using the reference inputs. The timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 (EV5) processor with a split primary cache (8 kB each of instruction and data cache), 96 kB of on-chip secondary cache, 2 MB of off-chip backup cache, and 512 MB of main memory, running Digital Unix/Alpha V4.0B (Rev. 564). In each case, the execution time reported is the smallest time of 10 runs.

4.2 Optimization of Constant Expressions

4.2.1 Interprocedural Constant Propagation, Constant Folding, and Strength Reduction

There are generally more opportunities for interprocedural constant propagation at (or after) link time than at compile time. There are three reasons for this:

1. The entire program, including all the library routines and an eventual runtime system, is available for inspection. Constants can be propagated across compilation unit boundaries and even source language boundaries.

2. Global data structures and subroutines have been placed within the programs address space by the linker. Hence their addresses are known constants at link time but unknown constants at compile time.

[1] We use statically linked executables because `Alto` relies on the presence of relocation information for the control flow graph construction. The Digital Unix/Alpha linker `ld` refuses to retain relocation information for non-statically-linked executables.

3. At link time some architecture-specific computations become available for optimization which are not visible at the intermediate code representation level typically used by compilers. An example of this case is the computation of the *gp* register on the Alpha processor: the value of this register is generally recomputed in the *init* node of a subroutine f as well as in *return* nodes following subroutine calls to ensure that it always carries the same value while code in f is executed [22]. In many cases, however, the recomputation is unnecessary and can be eliminated by propagating the value of the *gp* register through a program. It should be noted that this optimization cannot be carried out at compile time since the value of gp is only determined at link time.

The analysis used is based on the standard iterative constant propagation algorithm [2], limited to registers but carried out across the entire interprocedural control flow graph. This has the effect of communicating information about constant arguments passed in registers from a call site to the callee. To improve precision, we determine (by inspection) the registers saved on entry to a subroutine and restored at the exit from it: if a register *r* that is saved and restored by a subroutine in this manner contains a constant *c* just before the subroutine is called, then *r* is inferred to contain the value *c* on return from the call. [2] Our constant propagation is interprocedural and flow sensitive [52]. It is not context sensitive since data flow information from different call sites is not distinguished while being propagated through a subroutine. Context sensitive would require some notion of jump function [24] which would use up too much memory. Unlike [13] our analysis does not sacrifice precision in the presence of recursion, though. Similar to [76] our analysis is extended with the interpretation of conditionals. If a conditional

[2]Unfortunately, we cannot rely on the calling conventions being observed: hand-written assembly code in libraries does not always obey such conventions, and compilers may ignore them when doing interprocedural register allocation.

tests whether a register is equal to a constant, the constant will be propagated through the correct branch of the conditional. This simplifies the implementation of common case specialization (cf. Chapter 5).

As usual, constant propagation is interleaved with constant folding. The constant folder uses direct execution to compute the effect of the various opcodes (cf. Section 4.2.4).

It is noteworthy that even some load instructions can be “folded”. If we know the memory address an instruction loads from and this location belongs to a read-only section of the address space we can fetch the loaded value from the original executable.

Constant propagation is also interleaved with strength reduction. This might seem unnecessary at first, but strength reduction might change the control flow graph and hence might help in finding more constants. For example, after a subroutine start address has been propagated to an indirect call instruction (`jsr`), the callee becomes known and we no longer have to make worst case assumptions about it. [3]

[3]Due to some architectural peculiarities on the Alpha, initially most calls appear to be indirect calls

Program	Evaluable/All	Evaluated/All	Evaluated/Evaluable
`compress`	0.692	0.129	0.186
`gcc`	0.711	0.143	0.201
`go`	0.744	0.222	0.298
`ijpeg`	0.720	0.103	0.143
`li`	0.684	0.153	0.224
`m88ksim`	0.702	0.180	0.256
`perl`	0.715	0.167	0.234
`vortex`	0.700	0.241	0.344
Geom. Mean	0.708	0.162	0.228

Table 4.1: Effectiveness of Constant Propagation

The results of constant propagation are shown in Table 4.1. Column 2 lists the static number of instruction that produce a result (*evaluable* instructions) divided by all instructions. An example of an instructions that does not produce a result (*non-evaluable*) is a store instruction. Column 3 lists the static number of instructions whose result could be determined by Constant Propagation (*evaluated* instructions) divided by all instructions. Column 4 has the ratio of the previous columns.

The numbers were obtained after the second run of Constant Propagation during the base optimizations. This allows other optimizations, especially unreachable code elimination (cf. Section 4.3.4), to execute once, and hence makes the numbers more meaningful than the ones obtained after the first run. It can be seen that, on the average, it is possible to evaluate about 16% of the instructions of a program at link time. However, this does not mean that 16% of the instructions in a program can be eliminated. Some

of them may have side effects, such as control transfers, and so elimination is not possible. To eliminate others we would have to propagate the constant to all its uses and transform them into immediate operands there. On the Alpha this is only possible for small (8 bit) constants. Section 4.2.3 describes optimizations along this line. Often it is possible, though, to transform an instruction computing or loading a constant into a cheaper instruction (or instruction sequence) computing the same constant (cf. Section 4.2.2).

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	277.3	0.937
`gcc`	232.9	258.3	0.902
`go`	304.0	323.8	0.939
`ijpeg`	328.1	330.9	0.992
`li`	254.6	295.0	0.863
`m88ksim`	224.2	279.4	0.802
`perl`	182.0	220.5	0.825
`vortex`	316.4	444.4	0.712
Geometric Mean:			0.867

Table 4.2: Execution time impact of constant propagation

As shown in Table 4.2, this analysis has a profound impact on the performance of the generated code. For example, the SPECint95 benchmarks `li`, `m88ksim`, `perl`, and `vortex` suffer slowdowns of 15–30% when this analysis is turned off. The reason for this impact, in great part, is that many analyses and transformations rely on the knowledge of constant addresses computed in the program. For example, the code generated

by the compiler for a function call typically first loads the address of the called function into a register, then uses a `jsr` instruction to jump indirectly through that register. If constant propagation can be used to determine that the address being loaded is a fixed value, and the callee is not too far away, the indirect function call can be replaced by a direct call using a `bsr` instruction (this is a form of strength reduction): this is not only cheaper, but also vital for the construction of the interprocedural control flow graph of the program and for other optimizations such as inlining. Another example of the use of constant address information involves the identification of possible targets of indirect jumps through jump tables: unless this can be done, an indirect jump must be assumed as being capable of jumping to any node in the interprocedural control flow graph, [4] which can significantly hamper optimizations. Finally, knowledge of constant addresses is useful for alias analysis and the optimizations it enables, e.g., load and store avoidance.

4.2.2 Constant Generation

As described in the previous section it is often possible to determine, from constant propagation/folding, that a value being computed or loaded into a register is a constant. In such a situation the optimizer attempts to find a cheaper instruction to compute the same constant into that register. (This optimization could be generalized to cheap instruction sequences to replace high latency operations, such as integer multiplication.) The simplest case of this optimization involves computing the values of constants using specific registers whose values are known at each program point, namely, register $r31$, whose value is always 0, and the global pointer register *gp*, whose value at any program point is known at link time. If the (signed) constant k can be represented with 16 bits, the instruction to compute that constant into a register r is replaced by the instruction [5] `‘lda`

[4] More precisely, any basic block that is marked as “relocatable”. This is abstracted as the *unknownnode* (cf. Section 2.1.3).

[5] The meaning of the Alpha machine instructions is explained in Appendix A

`r, k(r31)'`. Similarly, if the difference between the constant *k* and the value of the *gp* register is representable as a signed 16 bit integer, we can do the same thing using *gp* as the base register. The basic optimization is described by Srivastava and Wall [68]; here it is generalized so that a constant can be computed from a known value in any register, not just *r*31 or *gp*. Furthermore, we are not limited to address constants.

Care must be taken to ensure that the constants involved are not *code addresses*, i.e. addresses pointing into code bearing parts of the Text Segment. Since our optimizations change the code, code addresses will change as well. Such constants are therefore excluded from this optimization. Other addresses, like data addresses, cause no problems, since the transformations implemented within our optimizer will leave them unchanged. To find out whether a constant might be a code address we use information from the Program Header. describing the structure of the segments , their start addresses, and their length. The answer will naturally be conservative, but so far we have found very few false positives in our benchmarks.

```
(1) ldq  r1, 16(gp)      (1) ldq  r1, 16(gp)      (1) ldq  r1, 16(gp)
(2) ldq  r2, 96(gp)      (2') lda  r2, 8(r1)      [(2') lda  r2, 8(r1)]
(3) ldq  r3, 32(gp)      (3') lda  r3, 16(r1)     [(3') lda  r3, 16(r1)]
(4) ldq  r4, 0(r1)       (4) ldq  r4, 0(r1)       (4) ldq  r4, 0(r1)
(5) ldq  r5, 0(r2)       (5) ldq  r5, 0(r2)       (5') ldq  r5, 8(r1)
(6) addq r4, r5,r6       (6) addq r4, r5, r6      (6) addq r4, r5, r6
(7) stq  r6, 0(r3)       (7) stq  r6, 0(r3)       (7') stq  r6, 16(r1)
```

(a) original code (b) after const. gen. opt. (c) after const. usa. opt.

Figure 4.2: Code generated for $a = b + c$

As an example of this optimization, consider the C statement "`a = b + c;`", where `a`, `b` and `c` are global (64 bit) variables of type `long`, with addresses 0x1400021558, 0x1400021560, and 0x1400021568. Figure 4.2 (a) shows the code generated for this statement by the compiler. Instructions $(1)-(3)$ load the addresses of the variables from the global address table, using the global pointer register, *gp*, to index into this table. Instructions $(4)-(7)$ implement the actual addition. The optimizer is able to determine the addresses loaded into registers *r*1, *r*2 and *r*3, since *gp* is constant within each subroutine and the global address table, it is pointing to, is a read-only area of memory. This allows constant value optimization of instructions (2) and (3), which replaces the address loads with cheaper `lda` instructions as shown in Figure 4.2 (b). Further optimizations are possible as described in the next section.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	278.4	0.933
`gcc`	232.9	239.2	0.974
`go`	304.0	308.7	0.985
`ijpeg`	328.1	327.6	1.001
`li`	254.6	271.3	0.939
`m88ksim`	224.2	248.2	0.903
`perl`	182.0	197.1	0.923
`vortex`	316.4	345.2	0.916
Geometric Mean:			0.946

Table 4.3: Execution time impact of constant generation

The performance impact of this optimization is illustrated in Table 4.3. The programs

that benefit the most from this optimization are `compress`, `li`, `m88ksim`, `perl`, and `vortex`, with improvements around 7%.

4.2.3 Constant Usage

Besides the generation of constants as results, we also attempt to optimize the use of constants as operands. Some Alpha instructions allow the use of a small immediate value in place of the second operand register. We exploit this feature whenever possible. If only the first operand register is determined to be constant, we try to swap the operands of the instruction. This is trivial if the instruction is commutative in its operands. If the instruction is not commutative, like a subtract instruction, we have two options. We can swap the operands and change the instruction opcode, i.e., we change the instruction which subtracts the second operand from the first to an instruction which subtracts the first operand from the second (reverse subtract). If this is not possible because such an instruction does not exist, we can still swap the operands but now we have to account for the fact that the instruction produces a different result. For example, in the case of a subtract instruction it produces the negative of the original value, and we must modify all uses of the result accordingly.

The optimizer also exploits the signed 16bit offsets in load and store instructions to make changes of the base register possible. The transition from Figure 4.2 (b) to (c) shows an example of this transformation. Instructions (5) and (7) are modified to use $r1$ as the new base register. This is compensated for by changing the offsets to make up for the difference in value between the original and the new base register. Note that registers $r2$ and $r3$ are no longer used in this code and will subsequently be deleted. Also note that this transformation might create internal pointers or hide other pointers and thus might conflict with a conservative garbage collector as described in [10]. It should therefore be turned off for those applications.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	260.1	0.999
`gcc`	232.9	236.9	0.983
`go`	304.0	303.9	1.000
`ijpeg`	328.1	329.4	0.996
`li`	254.6	255.4	0.997
`m88ksim`	224.2	226.8	0.989
`perl`	182.0	183.4	0.992
`vortex`	316.4	318.2	0.994
Geometric Mean:			0.994

Table 4.4: Execution time impact of constant usage

The performance impact of this optimization is illustrated in Table 4.4. The program that benefits the most from this optimization is `gcc` with a 1.7% improvement. For other programs the speedup is marginal.

4.2.4 Direct Execution

Constant folding is a difficult business. It requires us to provide an emulator for either an abstract machine (for constant folding in an ordinary compiler) or a concrete machine (for constant folding in a link time optimizer). Either way, emulating elementary operations using a high level language is very tedious and error prone. Often the high level language does not have equivalent operators for machine instructions, like a bitwise rotate instruction, and we need to resort to simpler bit manipulation operations provided by the language to emulate the rotate. Sometimes a language operator differs in subtle

ways from the equivalent operator provided by the machine or is unspecified, e.g., divide and modulo operations with negative operands. In the case of floating point operation the outcome is usually not even exactly defined, because only a certain number of digits are guaranteed to be accurate.

The constant folding portion of `gcc`, for example, consists of almost 5000 lines (150kB) of C source code [6].

In our implementation we have chosen a different route, and resorted to direct execution for constant folding [23]. In order to determine the result of an `addq` operation on two known values we actually execute an `addq` instructions. This guarantees perfect emulation of the machine behavior. It also requires very little programming effort (less than 100 lines) and is very fast. Of course, we have now made the optimizer quite non-portable but since there are other sources of non-portability this is only a minor sacrifice.

Our first implementation of the constant folder created a little subroutine each time we needed to fold a constant, i.e., at runtime. In case of the example above, the routine would consist of an `addq` and a `ret` instruction. The operand and destination registers of the `addq` instruction were chosen to mimic the calling conventions, so that the routine could be invoked from C using function pointers. Since we were reusing the same memory area for the little routine we had to make sure that the CPU would always "see" the latest snapshot. This was accomplished by invalidating the instruction cache before invoking the routine. This worked fine under Digital Unix/Alpha but caused problems under Linux/Alpha.

The most recent implementation avoids the instruction cache invalidation by generating a subroutine for each possible opcode once at the initialization of the optimizer.

Certain opcodes are excluded from direct execution, because they might raise exceptions (e.g., integer arithmetic instructions that trap on overflow), for other opcodes we

[6] file `fold-const.c` of `gcc` version 2.5.3

make sure that the arguments are valid, e.g., we do not execute floating point operations if one of the operands is a *not-a-number value* (NaN), like positive infinity.

4.3 Instruction Elimination

4.3.1 Useless Instruction Elimination

Our implementation of useless instruction elimination (also referred to as dead code elimination [2]) is solely based on register liveness information. If an instruction computes a value into register *r* and on all execution paths this register is not used before it is rewritten, we can in most cases eliminate the instruction. However, if the instruction has side effects we need to be more careful, e.g., if an instruction changes the flow of control besides computing a value, we cannot eliminate it. This rule has been relaxed for load instructions which always have the side effect of possibly causing a segmentation fault.[7]

Because we restrict our liveness analysis to registers, we will not detect useless computations whose value is stored into a memory cell from which it is never read. However, the store avoidance optimization described in Section 4.3.3 will catch a few of these cases.

[7] Curiously enough, this caused problems with an early version of Boehm's conservative garbage collector [10], which used a useless load instruction to probe for the boundaries of the address space.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	262.9	0.988
`gcc`	232.9	241.1	0.966
`go`	304.0	309.4	0.982
`ijpeg`	328.1	329.5	0.996
`li`	254.6	265.1	0.960
`m88ksim`	224.2	238.8	0.939
`perl`	182.0	193.9	0.938
`vortex`	316.4	341.2	0.927
Geometric Mean:			0.962

Table 4.5: Execution time impact of useless instruction elimination

The performance impact of this optimization is illustrated in Table 4.5. The programs that benefit the most from this optimization are `gcc`, `li`, `m88ksim`, `perl`, and `vortex`, with improvements around 5%.

4.3.2 Move Elimination

The aim of the move elimination optimization is identical to copy propagation, viz., we try to reduce the number of move instructions. However, our optimization is more general and goal directed. The move elimination optimization examines each move instruction of the program in turn and tries to eliminate it by register renaming. This is done locally within a basic block only, using register liveness information. Move elimination considers three patterns:

1. `move  ra,rb     => [move ra,ra]`
   `...                ...`

```
use   rb        => use ra
...                ...
lastuse rb      => use ra
```

In this pattern the last use of the target (*rb*) of the move instruction is in the same basic block as the move instruction and the source (*ra*) of the move instruction is still available at the last use of the target. Hence we can convert all uses of the target into uses of the source and eliminate the move instruction. Note that this is the only pattern where copy propagation would yield the same effect. Patterns 2 and 3 cannot be handled by copy propagation.

This pattern will also work for `addq` and similar instructions instead of move instructions, if one of the operands is a constant and this constant can be combined into the uses. For example:

```
addq  ra,2,rb        => [eliminated]
...                 ...
ldq rc,x(rb)         => ldq rc,x+2(ra)
...                 ...
addq  rb,6,rc        => addq ra,8,rc
```

Note, that this extension might create internal pointers or hide other pointers and thus might conflict with a conservative garbage collector as described in [10]. It should therefore be turned off for those applications.

2.
```
def   ra        => def rb
...
move  ra,rb     => [move rb,rb]
...
```

```
lastuse ra      => use rb
```

In this pattern the definition of the source of the move (*ra*) instruction and the last use of *ra* are in the same basic block as the move instruction. The target (*rb*) of the move instruction is not live between the definition and the move instruction. Also, *rb* must still be available at the last use of *ra*. Hence we can convert all uses of the target into uses of the source and eliminate the move instruction.

3.
```
def   ra        => def rc
...
move  ra,rb     => [move rc,rc]
...
use   rb        => use rc
...                ...
lastuse rb      => use rc
...
lastuse ra      => use rc
```

In this pattern the entire live range of the source (*ra*) and the target (*rb*) of the move instruction are located in the same basic block. If we can find a scratch register *rc* which is available from the definition of *ra* to the end of both the live ranges of *ra* and *rb* we can rename all uses and definitions of *ra* and *rb* into uses and definitions of *rc*.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	261.2	0.995
`gcc`	232.9	237.6	0.980
`go`	304.0	306.0	0.993
`ijpeg`	328.1	329.3	0.996
`li`	254.6	256.3	0.993
`m88ksim`	224.2	227.1	0.987
`perl`	182.0	182.2	0.999
`vortex`	316.4	316.6	0.999
Geometric Mean:			0.993

Table 4.6: Execution time impact of move elimination

The performance impact of this optimization is illustrated in Table 4.4. The program that benefits the most from this optimization is `gcc` with a 2.0% improvement. For other programs the speedup is marginal

4.3.3 Load and Store Avoidance

It is sometimes possible to identify load and store operations as unnecessary. Suppose that an instruction I_1 stores a register r_1 to memory region l (or loads r_1 from memory region l), and is followed soon after by an instruction I_2 that loads from location l into register r_2. If it can be verified that that location l is not modified between these two instructions, then *load avoidance* attempts to delete instruction I_2 and replace it with a register move from r_1 to r_2. It may happen that register r_1 is overwritten between instructions I_1 and I_2: in this case, the optimizer tries to find a free register r_3 that can be used to hold the value in r_1. If the instruction I_1 can now be shown to be dead, it can be

deleted.

A similar optimization can be applied to two store instructions I_1 and I_2 following each other and accessing the same memory region. The first one is clearly useless and can be deleted. Note that since our useless code elimination optimization focuses on registers and ignores memory regions, it will not catch this case. We also employ a very basic liveness analysis for stack locations to eliminate useless stores to the stack.

Optimization opportunities like the ones described above can be easily exploited by a compiler. However, we encounter additional opportunities at link time for a variety of reasons: a variable may not have been kept in a register by the compiler because it is a global, or because the compiler was unable to resolve aliasing adequately, or because there were not enough free registers available to the compiler. At link time, accesses to globals from different modules become evident, making it possible to keep them in registers [74]. Inlining across module boundaries and inlining of library routines may make it possible to resolve aliasing beyond what can be done at compile time. A link time optimizer may be able to scavenge registers that can be used to hold values that were spilled to memory by the compiler. Finally, code restructuring transformations, such as basic block duplication, might convert a partially redundant load into a fully redundant load.

Many memory accesses result from the saving and restoring of callee-save registers at subroutine boundaries. Some of these accesses may be unnecessary, either because the registers saved and restored in this manner are not touched along all execution paths through a subroutine, or because the code that used those registers became unreachable, e.g., because the outcome of a conditional branch could be predicted as a result of inlining or interprocedural constant propagation, and therefore was deleted. To reduce the number of such unnecessary memory accesses, the optimizer uses a variation on *shrink-wrapping* [16] to move register save/restore actions away from execution paths that do

not need them. The difference between our implementation of shrink-wrapping, and that originally proposed by Chow [16], is that we do not allow any execution path through a function to contain more than one save and restore operation for a particular register. Apart from this, if a function saves and subsequently restores a callee-save register *r* but does not change *r*, the instructions to save and restore *r* are eliminated.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	261.0	0.995
`gcc`	232.9	238.9	0.975
`go`	304.0	304.4	0.998
`ijpeg`	328.1	328.1	1.000
`li`	254.6	258.1	0.987
`m88ksim`	224.2	228.2	0.983
`perl`	182.0	186.1	0.978
`vortex`	316.4	315.8	1.002
Geometric Mean:			0.990

Table 4.7: Execution time impact of load and store avoidance

The performance impact of the load and store avoidance optimization is illustrated in Table 4.7. The programs that benefit the most from this optimization are `gcc`, `li`, and `perl`, with improvements ranging around 2%.

4.3.4 Unreachable Code Elimination

Unreachable code —i.e., code that will never be executed — typically arises at compile time due to user constructs (such as debugging statements that are turned off by setting

a flag) or as a result of other optimizations, and is usually detected and eliminated using intra-procedural analysis. By contrast, unreachable code that is detected at link time usually has very different origins: most of it is due to the inclusion of irrelevant library routines, together with some code that can be identified as unreachable due to the propagation of actual parameter values into a subroutine. In either case, link time identification of unreachable code is fundamentally interprocedural in nature.

Even though unreachable code can never be executed, its elimination is desirable for a number of reasons:

1. It reduces the amount of code that the optimizer needs to process, and can lead to significant improvements in the amount of time and memory used.

2. It can enable optimizations that otherwise might not be possible, such as bringing two basic blocks closer together, allowing for more efficient control transfer instructions to be used, or allowing for a more precise liveness analysis which might trigger several other optimizations.

3. It can reduce the amount of "cache pollution" caused by unreachable code that is loaded into the cache when nearby reachable code is executed. This in turn can improve the overall cache behavior of the program.

4. It simplifies certain analyses because after unreachable code elimination we can assume that every node is reachable from a subroutine *init* node or the *unknownnode*.

Unreachable code analysis involves a straightforward depth-first traversal of the (interprocedural) control flow graph, and is performed as soon as the (interprocedural) control flow graph of the program has been computed, and is repeated later with the base optmizations. Initially, all nodes are marked as dead, and then nodes are marked reachable only if they can be reached from another node that is reachable. The entry point of the

program (the *init* node of *entryfun*) is always reachable. We ignore *return* edges, hence a *return* node will be marked reachable only if the corresponding *call* node is reachable.

Program	orig. code (kB)	unreachable code (kB)	ratio
`compress`	103472	21396	0.207
`gcc`	1503376	94504	0.063
`go`	363392	28312	0.078
`ijpeg`	302784	54592	0.180
`li`	188752	37000	0.196
`m88ksim`	244960	39256	0.160
`perl`	474000	43904	0.093
`vortex`	758928	139772	0.184
Geometric Mean:			0.133

Table 4.8: Effectiveness of unreachable code elimination

Due to technical reasons it is currently not possible to disable unreachable code elimination without disabling other optimization in our optimizer, hence we only report static improvements on the code size. The amount of unreachable code detected in our benchmarks is shown in Table 4.8. It can be seen that the amount of unreachable code is quite significant: in most programs, it exceeds 15%. On the average, about 13% of the instructions in our benchmarks were found to be unreachable. This is somewhat higher than the results of Srivastava, whose estimate of the amount of unreachable code in C and Fortran programs was about 4%–6% [67].

4.4 Code Motion and Restructuring Optimization

While the optimizations described in the previous sections are always beneficial and should be applied whenever possible, the optimizations presented next are best performed when guided by execution frequency profiles.

4.4.1 Inlining

Inlining replaces a call to a subroutine with a copy of its body. It can be a very useful optimization, because it eliminates the overhead associated with the call and allows us to specialize the body for a particular call site (calling context). However, inlining is a two edged sword. Many people have found unexpected performance degradation when experimenting with inlining.

- Inlining at the source level might increase the register pressure [26] and lead to suboptimal register allocations.
- In FORTRAN programs, the compiler might no be able to exploit the no-alias requirement for arguments of subroutine calls, once such a subroutine has been inlined [20].
- Inlining of recursive subroutines may lead to stack explosion [14].
- Through the increase in code size inlining might hurt instruction cache performance [53].

Our optimizer inlines subroutines at the object code level. This avoids the problems with the increased register pressure and FORTRAN calling conventions, but is somewhat more complex than inlining at the source level, where inlining is just a syntactical transformation. If a subroutine contains a computed jump, for example, it is not sufficient to

merely clone the subroutine body and insert it at the call site. We also need to clone the jump table. In order to deal with the problem of increased code size we employ execution frequency profiles.

The optimizer inlines a subroutine if

- it is small (less than 4 instructions).
- it has only one call site.
- it is called very frequently from a call site.

The first two cases are always beneficial since there is no increase in code size. In the last case we reduce call overhead at the expense of code growth. We do not take the benefits of call site specialization into account when making inlining decisions.

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	269.6	0.964
`gcc`	232.9	237.6	0.980
`go`	304.0	304.1	0.999
`ijpeg`	328.1	328.5	0.999
`li`	254.6	259.7	0.980
`m88ksim`	224.2	237.4	0.944
`perl`	182.0	178.2	1.021
`vortex`	316.4	321.3	0.985
Geometric Mean:			0.984

Table 4.9: Execution time impact of inlining

The performance impact of this optimization is illustrated in Table 4.9. The programs that benefit the most from this optimization are `compress`, `gcc`, `li`, `m88ksim`, with improvements ranging around 3%. The `perl` benchmark suffers a slowdown of 2.1% indicating that some more fine tuning of the optimization is necessary.

4.4.2 Code Positioning

Program	with opt.(sec)	without opt.(sec)	with/without
`compress`	259.8	259.1	1.003
`gcc`	232.9	264.5	0.881
`go`	304.0	309.7	0.981
`ijpeg`	328.1	327.9	1.000
`li`	254.6	260.9	0.976
`m88ksim`	224.2	274.1	0.818
`perl`	182.0	204.5	0.890
`vortex`	316.4	372.2	0.850
Geometric Mean:			0.922

Table 4.10: Execution time impact of code positioning

Our code positioning is a variation of the Pettis-Hensen algorithm [57]. The algorithm uses execution counts of control flow edges to achieve two goals:

- Minimization of the dynamic count of control flow changes (taken branches):
  This is achieved by rearranging the basic blocks so that if basic block *A* most likely transfers control to basic block *B*, then *B* follows *A* in the program text. Decreasing control flow changes improves the performance of pipelined CPUs like the Alpha.

- Minimization of instruction cache misses:

  This is achieved by grouping pieces of code that are likely to execute shortly after another close together, thereby reducing the probability that they cause a conflict in the instruction cache. Instruction caches usually have small associativity so that more conflicts translate to more misses. Moving less frequently executed code away from frequently executed code reduces instruction cache pollution and improves instruction cache utilization which leads to a reduction of capacity misses.

The performance impact of this optimization is illustrated in Table 4.10. As observed by others [17] before, this optimization yields substantial speedups. The programs that benefit the most from this optimization are `gcc`, `li`, `m88ksim`, `perl`, and `vortex`, with improvements ranging from 5% to 16%.

4.5 Overall Effectiveness

In this section we measure the overall effectiveness of the optimizer and compare it with the vendor supplied optimizers.

4.5.1 Without Profiles

Table 4.11 compares the execution times of the SPECint95 benchmarks when compiled with the vendor supplied C compiler with and without an additional run of the `Alto` based link time optimizer. No profiling information is used.

The C compiler was invoked as

```
cc -O4 $(CFILES) -non_shared -WL,-z -WL,-d -WL,-r
-lm -o exe.cc
```

where `CFILES` is a list of all the C source files for the program. The resulting executable was optimized using

```
alto -i exe.cc -o exe.alto
```

The optimizer achieves an average speedup of 13.9%.

Program	cc (sec)	Alto (sec)	Alto/cc
compress	282.1	263.9	0.935
gcc	290.2	259.5	0.894
go	346.5	306.2	0.884
ijpeg	337.7	329.7	0.976
li	315.5	262.4	0.832
m88ksim	337.0	261.7	0.777
perl	247.9	209.2	0.844
vortex	493.4	378.1	0.766
Geometric Mean:			0.861

Table 4.11: Overall execution time impact (without profiles)

4.5.2 With Profiles

Next, we measured the performance achievable using the existing capabilities for static optimization available under Digital Unix/Alpha. For this, we compiled the benchmarks at the same optimization level as before, but additionally with profile-directed inter-file optimization and link time optimization using OM [67]. For this, the programs were compiled as follows:

1. First, the benchmarks were compiled as

```
cc -O4 $(CFILES) -non_shared -lm -o exe.cc
```

where `CFILES` is a list of all the C source files for the program.

2. The resulting executable `exe.cc` was instrumented with `pixie` and run on the SPEC95 training input for the benchmark to produce an execution profile. A feedback file was then generated from this profile using the command

```
prof -pixie -feedback fb.cc exe.cc
```

3. The source files were recompiled with profile-guided and inter-file optimization turned on, using the feedback file generated in the previous step:

```
cc -O4 -ifo -inline speed -feedback fb.cc $(CFILES)
   -non_shared -lm -o exe.ccfb
```

The switch `-ifo` turns on inter-file optimization and `-inline speed` instructs the compiler to inline routines to enhance execution speed.

4. The resulting executable `exe.ccfb` was again instrumented with `pixie`, using the SPEC95 training inputs.

5. The resulting profiling information for `exe.ccfb` was used to recompile the program a final time, this time with the `OM` link time optimizer turned on as well:

```
cc -O4 -ifo -inline speed -feedback fb.ccfb -om
   -WL,-om_compress_lita -WL,-om_dead_code
   -WL,-om_ireorg_feedback,exe.ccfb
   $(CFILES) -non_shared -lm -o exe.final
```

The reason it is necessary to regenerate the profile information for `OM` is that the feedback-directed optimizations can change code addresses, rendering the original profile useless from the perspective of `OM`. Notice that in this step, two distinct

sets of profiles are being used: the feedback file `fb.cc`, generated from the original profile obtained in step 2; and the profile for `exe.ccfb`, obtained for the executable resulting from feedback-directed inter-file optimization in step 4.

Compared to the procedure just described, optimizing a benchmark utilizing profiling information with the `Alto` based optimizer is rather simple.

1. First, the programs were compiled as

   ```
   cc -O4 $(CFILES)-non_shared -WL,-z -WL,-d -WL,-r
   -lm -o exe.cc
   ```

   where `CFILES` is a list of all the C source files for the program.

2. The resulting executable `exe.cc` was instrumented with `pixie` and run on the SPEC95 training input for the benchmark to produce an execution profile.

3. Finally the `Alto` based optimizer was run exploiting the profiling information generated in the previous step

   ```
   alto -i exe.cc -o exe.final
   ```

Table 4.12 is similar to Table 4.11 except that we allow both the vendor supplied compiler and the `Alto` based optimizer to utilize profiling information obtained from training input of the SPECint95 benchmarks.

As can be seen from Table 4.12, the `Alto` based optimizer beats the best optimization techniques provided by the vendor by 5.7%.

Program	`cc` (sec)	`Alto` (sec)	`Alto`/`cc`
`compress`	272.8	259.8	0.952
`gcc`	226.3	232.9	1.029
`go`	299.7	304.0	1.014
`ijpeg`	332.9	328.1	0.986
`li`	288.2	254.6	0.883
`m88ksim`	230.8	224.2	0.971
`perl`	201.7	182.0	0.902
`vortex`	390.3	316.4	0.811
Geometric Mean:			0.941

Table 4.12: Overall execution time impact (with profiles)

CHAPTER 5

COMMON CASE SPECIALIZATION

In the previous chapter we discussed classical compiler optimizations in the context of link time optimization. Only a few of the optimizations exploited profiling information, and those that did used simple execution frequency profiles. In this chapter we describe a highly speculative optimization exploiting additional, more sophisticated profiling information. Even though, this optimization can be incorporated into a regular compiler, it is presented here in the context of link time optimization.

Knowledge that an expression in a program can be guaranteed to evaluate to some particular constant at compile time can be profitably exploited by constant folding (cf. Section 4.2.1). This is an "all-or-nothing" transformation, in the sense that unless the optimizer is able to guarantee that the expression under consideration evaluates to a compile time constant, the transformation cannot be applied. In practice, however, it is often the case that an expression at a point in a program "almost always" takes on a particular value [11]. As an example, in the SPEC95 benchmark `perl`, the function `memmove()` is called close to 24 million times: in almost every case, the argument giving the size of the memory region to be processed has the value 1; we can take advantage of this fact to direct such calls to an optimized version of the function that is significantly simpler and faster. As another example, in the SPEC95 benchmark `li`, a very frequently called function, `livecar()`, contains a `switch` statement where one of the case labels, corresponding to the type `LIST`, occurs over 80% of the time; knowledge of this fact allows the code to be restructured so that this common case can be tested first, and so control

does not have to go through the jump table, which is relatively expensive. As these examples suggest, if we know that certain values occur very frequently at certain points in a program, we may be able to take advantage of this information to improve the performance of the program. Unfortunately, classical compiler techniques cannot be used to take advantage of knowledge of the distribution of values in order to optimize for the common case. The idea behind common case specialization is to employ sophisticated (value) profiles to allow such optimization.

There are a number of technical issues that have to be addressed to accomplish this. Specializing a piece of code for "too many" different values, or specializing code where the benefits of specialization are not high enough, can lead to performance degradation. It is therefore necessary to determine what code to specialize, and to what extent. It is also necessary to determine how the specialization should be carried out, so that the common case is made efficient while the code remains correct.

The following sections address these questions and show how value-profile-based specialization can be automated and integrated into the link time optimizer presented in the previous chapter.

Related work: There is a considerable body of work on program specialization within the partial evaluation community: Jones *et al.* give an extensive discussion and bibliography [43]. This work focuses largely on aggressive code specialization starting with known values for some or all of a program's inputs. The issue of specialization based on value profiles is not considered.

Some implementations of object-oriented languages attempt to mitigate the high cost of dynamically dispatched calls using a limited form of value-profile-based specialization. The idea — referred to as *type feedback* or *receiver class prediction* [3, 41] — is to monitor the targets of dynamically dispatched function calls, and to use this information to inline the code for frequently called targets. The main limitation of this approach

is that the specialization is restricted to dynamically dispatched function calls, and so will not be applied to "ordinary" code even if such code could benefit substantially from knowledge of the values most commonly encountered at runtime.

Another approach, termed *dynamic compilation*, specializes code at run time [5, 50, 30, 19]. It focuses on values which are unknown at compile time but constant at run time. Those values are usually identified with support from the programmer, through source code annotations, and the process is therefore not fully automatic. The optimization is usually performed by producing a machine code template at compile time and then filling in the blanks at run time: this causes additional overhead. Furthermore, this approach implies that the code may not be optimized to the fullest extent, since the template is not specialized for each filled in value.

Calder *et al.* have investigated issues and techniques for value profiling extensively [11]. Our implementation of value profiling was inspired by theirs and is very similar to it. While Calder *et al.* consider profiling both registers and memory locations, we profile only registers. We use a two-stage profiling process in order to reduce the time and space overheads. The idea is to first profile the application using a simple basic-block profiler such as `pixie`, and then use the execution frequency information so obtained to identify candidates for value profiling and specialization. In a different paper, Calder *et al.* discuss value-profile-based optimization [12]; they use hand-transformed examples to show that value-profile-based specialization can yield significant speed improvements. By contrast, the work presented here describes value-profile-based specialization that is fully automatic and that has been integrated into the `Alto` system. This automation is nontrivial, since it requires a careful cost-benefit analysis within the optimizer to avoid degradation of performance. The details of this cost-benefit analysis, and of how the specialization is carried out, are described in the following sections.

5.1 Preliminaries

Suppose we have a code fragment C that we wish to specialize for a particular value v of a register (or variable) `r`. Conceptually, value-profile-based specialization transforms the code C into:

if (r == v) **then** $\langle C \rangle_{\mathtt{r}=v}$ **else** C

where $\langle C \rangle_{\mathtt{r}=v}$ represents the residual code of C after it has been specialized to the value v of r. The test '**if** `(r == v)` ... ' is needed because we cannot guarantee that r will always take on the value v at that program point. The idea can be generalized to multiple values: given a probability distribution on these values, we can use a collection of tests such as the one above, organized as an optimal binary search tree, to choose between the specialized versions. For simplicity of discussion, we focus on specialization for a single value here, since this illustrates the technical issues that arise.

Notice that, while the specialized code $\langle C \rangle_{\mathtt{r}=v}$ may be more efficient than the original code C, the overall transformed code will actually be less efficient than the original code for values of r other than v, since a runtime test has been introduced. There is thus a tradeoff associated with the transformation: the cost of some execution paths may be reduced by the specialization, but this will be accompanied by an increase in the cost of other execution paths. If this tradeoff is not assessed carefully, specialization can lead to a degradation in performance.

Before we can actually carry out any code specialization, there are a number of decisions that have to be made. Specifically, we have to decide the program point[1] p where the specialization should begin (this corresponds to the point where runtime tests on values have to be inserted, as discussed above); the register r whose values we are interested

[1] For our purposes a "program point" refers to the points immediately before or after an instruction; this includes the entry and exit points of basic blocks.

in;[2] the particular value(s) *v* that we specialize for; and the actual code fragment *C* that is to be subjected to specialization. A *specialization triple* is a triple of the form (p, r, v), where *p* is a program point, *r* is a register, and *v* is a value for that register; such triples identify the runtime tests that have to be inserted in the context of value-profile-based specialization and the program points where they must be inserted. The *specialization region* of a triple (p, r, v) refers to the region of code that is chosen for specialization; this identifies the code fragments that appear in the **then** and **else** branches of the runtime test corresponding to that triple. The details of how specialization triples and regions are chosen are discussed in the next section.

5.2 Code Specialization

Value-profile-based code specialization is a three-step process. Section 5.2.1 describes a cost-benefit analysis that is fundamental to our approach. In order to reduce the time and space overheads of value profiling as far as possible, we first identify which (*program point, register*) pairs merit profiling. Section 5.2.2 discusses how this is done so as to avoid profiling instructions that cannot help us speed up the program. The second step, discussed in Section 5.2.3, is to carry out the instrumentation and profiling itself so as to obtain the value profiles. The final step, discussed in Section 5.2.4, uses the value profiles to carry out specialization for those program points where this is deemed to be profitable.

[2] In general, specialization can be carried out based on the value of a register, variable, or memory location, or relationships between such values. To simplify the discussion, and because our current implementation carries out specialization based on register values, we refer to register values when discussing specialization.

5.2.1 Estimating Costs and Benefits of Specialization

Our value profiling and specialization decisions are guided by estimates of the benefit that would be obtained from code specialization, given the knowledge that a register r has value v at a program point p, denoted by $\mathsf{Benefit}(p,r,v)$. The benefit tries to approximate the number of saved CPU cycles. As we will explain below the estimate gives a lower bound on the actual benefit. There are two components to the computation of benefits:

(i) For each instruction I that uses the value of r available at p, there may be some benefit to knowing that its value is v. The magnitude of this benefit will, in general, depend on the type of I, the operand position where r occurs, and the actual value v of r, and is denoted by $\mathsf{Savings}(I,r,v)$.

(ii) It may happen that knowing the value of an operand register r of an instruction I allows us to determine the value computed by I. In this case, I is said to be *evaluable* given r, written $\mathsf{Evaluable}(I,r)$. If I is evaluable given r, the benefits obtained from specializing other instructions that use the value computed by I for a particular value of r can also be credited to knowing the value of r at p. The indirect benefits so obtained from knowing that the value of r in instruction I is v are denoted by $\mathsf{IndirBenefit}(I,r,v)$.

Let $\mathsf{ExeFreq}(I)$ denote how often I is executed. Let $\mathsf{Uses}(p,r)$ denote the set of all instructions that use the value of register r that is available at program point p. Then the benefit of knowing that a register r has value v at program point p is given by the following:

$$\mathsf{Benefit}(p,r,v) = \sum_{I \in \mathsf{Uses}(p,r)} (\mathsf{ExeFreq}(I) \times \mathsf{Savings}(I,r,v) + \mathsf{IndirBenefit}(I,r,v))$$

The indirect benefits of knowing that register r has value v at instruction I is given by the following. Here, p' is the program point immediately after I; $\mathit{ResultReg}(I)$ denotes the

register into which I computes its result; and $\mathit{Value}_{r=v}(I)$ denotes the value computed by instruction I given that register r has the value v (this is undefined if $\mathsf{Evaluable}(I,r)$ is false).

$$\mathsf{IndirBenefit}(I,r,v) = \begin{cases} \mathsf{Benefit}(p', \mathit{ResultReg}(I), \mathit{Value}_{r=v}(I)) & \text{if } \mathsf{Evaluable}(I,r) \\ 0 & \text{otherwise} \end{cases}$$

An approximation is made when estimating the indirect benefit, because the fact that r equals v is forgotten in the case that I becomes evaluable. The only information that is propagated in that case is $\mathit{ResultReg}(I)$ equals $\mathit{Value}_{r=v}(I)$. Consider the following code example:

```
ldq  r1, 4(r6)    # load value from memory into r1
addq r1, 1, r2    # compute
mulq r1, r2, r0   #   r1 * (1+r1) into r0
```

We want to compute the benefit of knowing $r1$'s value after the load instruction. Knowing $r1$ will make the `addq` instruction evaluable, hence adding to the overall benefit and making $r2$ also known. However, the `mulq` instruction will not appear to be evaluable, we just obtain the sum of the benefits of knowing $r1$ and $r2$ separately. However, such case are rare and the approximation allows us to simplify the implementation drastically: The equations for computing benefits propagate information from the uses of a register to its definitions. These equations can, in general, be recursive, corresponding to a cycle in the def-use chain. The standard approach to solving recursive equations in the context of compile-time program analysis is to compute—usually iteratively—a fixpoint over a suitable domain. We do not follow this approach, instead we use our cycle-free use-def chain datastructure (cf. Section 3.2) and propagate information bottom-up from uses of registers to their definitions, in a single pass.

The benefits of specialization have to be weighed against the costs incurred due to runtime tests. The cost of such a test depends on the register and value being tested: e.g., testing for the value 0 is usually fairly cheap, while testing for a non-zero floating point constant may incur a load from memory. The cost of testing whether a register r has a value v is denoted by $\mathsf{TestCost}(r,v)$. The cost tries to approximate the number of additional CPU cycles needed.

5.2.2 Identifying Candidates for Specialization

In order to reduce the time and space overheads for value profiling as far as possible, we attempt to identify candidate (*program point*, *register*) pairs for which specialization could conceivably yield a performance improvement if we had a sufficiently skewed runtime distribution of values. This is done by estimating, for each such pair, the maximum benefit $\mathsf{MaxBenefit}(p,r)$ that could be achieved via specialization if the value of r at p was completely invariant dynamically—that is, always had the same value. As in the case of benefits, discussed in the previous section, the computation of this quantity has two components. The maximum savings incurred from the specialization of an instruction I to the value of a register r is given by

$$\mathsf{MaxSavings}(I,r) = \max_{v} \mathsf{Savings}(I,r,v)$$

In the implementation, of course, we do not compare the values of $\mathsf{Savings}(I,r,v)$ for all possible v, but resort to what is essentially a table lookup. The maximum benefit is then given by

$$\mathsf{MaxBenefit}(p,r) = \sum_{I \in \mathsf{Uses}(p,r)} (\mathsf{ExeFreq}(I) \times \mathsf{MaxSavings}(I,r) + \mathsf{MaxIndirBenefit}(I,r))$$

where $\mathsf{MaxIndirBenefit}(I,r)$ is given by

$$\mathsf{MaxIndirBenefit}(I,r) = \begin{cases} \mathsf{MaxBenefit}(p', \mathit{ResultReg}(I)) & \text{if } \mathsf{Evaluable}(I,r) \\ 0 & \text{otherwise} \end{cases}$$

Thus, $\mathsf{MaxBenefit}(I,r)$ takes into account the type of the instruction and operand positions where *r* occurs, but not the actual value of *r* at *p*, since this is not yet known. Thus, the division instruction '`z = x div r`' will be given a greater benefit than '`z = x div r`', because the first case offers greater possibilities for strength reductions based on the knowledge of the value of *r*. On the other hand, '`z = r div 32`' is evaluable and therefore is given an even higher benefit which is strongly related to the latency of the `div` instruction. Conditional branches are never evaluable since they do not compute a value; however, these instructions are assigned a relatively high benefit, since the branch can be optimized away if the condition register's value is known.

Analogously, the minimum cost of testing a register *r* for a value *v* is given by

$$\mathsf{MinTestCost}(r) = \min_{v} \mathsf{TestCost}(r,v).$$

Once maximum benefits have been computed as described above, the candidates for profiling are chosen as follows: register *r* is value-profiled at point *p* if and only if

$$\mathsf{MaxBenefit}(p,r) > \mathsf{MinTestCost}(r) \times \mathsf{ExeFreq}(p).$$

5.2.3 Value Profiling

Given a set of (*program point*, *register*) pairs to be value-profiled, we use a scheme based on that of Calder *et al.* [11] for obtaining value profiles. As mentioned earlier, our implementation of value profiling obtains profiles for registers only, not for memory locations. The actual profiling is carried out by a function created for this purpose. This function compares the value of the register in question with the contents of a fixed-size

table of previously encountered values. If the current value is already in the table, the count of that value is incremented; otherwise, if the table is not full, the value is added to the table and its count initialized to 1. If the table is full the value is ignored. Periodically, the table is cleaned by evicting the least frequently used values from the table: this allows new values to enter the table. We also keep track of the total number of times execution passes through the point p by incrementing a counter associated with that point.

We have also implemented a variant of this scheme that we call *predicate profiling*, where we ask how often a given predicate is satisfied at a given program point. Examples of such predicates are: "is the value of a given register r non-negative?" or "is the value of register r_a different from that of register r_b?" Notice that predicate profiles are not simply summaries of value profiles: e.g., given value profiles for registers r_a and r_b, we cannot in general reconstruct how often the predicate $r_a == r_b$ holds. The predicate that we choose to profile at any program point is typically determined by considerations of the possible optimizations that might be enabled based on that predicate's profile. Predicate profiles are important for three reasons. First, they conceptually generalize the notion of value profiles by allowing us to capture the distribution of relationships between different program entities. Second, a predicate profile may have a skewed distribution, and therefore enable optimizations, even if the value profiles for the constituents of the predicate profile are not very skewed: for example, a predicate $r_a \neq r_b$ may be true almost all of the time even if the values in r_a and r_b do not have a very skewed distribution. Finally, the implementation of predicate profiling can be made more efficient than that of general value profiling because we know that the evaluation of a predicate can take on at most two values.

5.2.4 Carrying out the Specialization

Once the value profiles have been obtained, code specialization involves two steps. First, it is necessary to determine the particular specialization triples that should be considered, and the corresponding specialization regions. The code transformation is then carried out by cloning the specialization region, incorporating the clone into the code together with tests on register values as described above, and carrying out the actual specialization.

5.2.4.1 Identifying Specialization Triples

The benefit computation described in Section 5.2.1 is used to identify the specialization triples for which code specialization is worthwhile. Since we know the distribution of the values taken on at the points that have been profiled, we can determine the probability $\mathsf{prob}(v)$ with which a value v occurs. Specializing at a program point p for a value v of a register r is then worthwhile only if

$$\mathsf{Benefit}(p,r,v) \times \mathsf{prob}(v) > \mathsf{TestCost}(r,v) \times \mathsf{ExeFreq}(p)$$

Once we have identified the set of specialization triples for which the benefits of specialization exceed its runtime cost, we have to choose which of these should actually be specialized. An issue that has to be addressed here is that the specialization regions for different such triples may overlap. This is illustrated by the following instruction sequence:

```
ldq r5, 0(r4)            # r5 := load from 0(r4)
and r5, 0xff, r6         # r6 := r5 & 0xff
```

Suppose that we have value profiled register $r5$ after the `ldq` instruction and register $r6$ after the `and` instruction, and that based on the cost benefit analysis, both of these instructions are candidates for specialization. However, the program points are dependent—$r6$

is computed from $r5$—and their specialization regions overlap. Depending on the circumstances, it might be better to specialize based on the `ldq` instruction because more instructions use the result of this instruction; in other situations, it might be better to specialize based on the `and` instruction because its value distribution might be more skewed. In such cases, we specialize only the more promising one, based on the cost benefit analysis; in the case of a tie, the program point that dominates the other is chosen (as discussed in Section 5.2.4.2, overlaps are not possible unless one of the points dominates the other).

5.2.4.2 Identifying Specialization Regions

Given a set of specialization triples, we have to determine the specialization region associated with each of them. The basic intuition is that given a triple (p, r, v), we want to identify the instructions that, directly or indirectly, use the value of r available at p, and so might potentially benefit from specialization (cf. Figure 5.1).

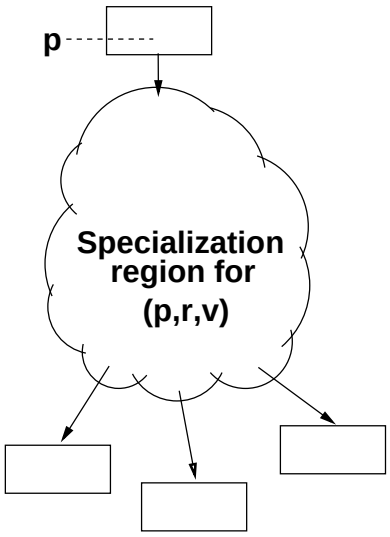


Figure 5.1: Specialization region

We first make precise the notion of an instruction using a value "directly or indirectly." Given a program point p and register r, we say that (p,r) *influences* an instruction I if (i) I uses the value of r at p; or (ii) there is an instruction J at a program point p' such that J defines a register r', (p,r) influences J, and (p',r') influences I. Then, given a triple (p,r,v), the specialization region for this triple is defined to be the smallest set of basic blocks R such that

- the basic block B_p containing p is in R;
- if (p,r) influences an instruction I occurring in a basic block B_I, and p dominates B_I, then B_I is in R; and
- if B is in R, $B \neq B_p$, and B' is a (immediate) intra-procedural predecessor of B in the control flow graph of the program, then B' is in R.

It is not hard to see that, given a specialization triple (p,r,v), the basic block B_p containing p dominates every block in the specialization region of this triple. This is necessary for correctness: we have to ensure that any execution path that can reach the specialization region of this triple must pass through the test inserted at p.

There are two issues that are not addressed by this definition of specialization regions. The first is that, given a triple (p,r,v), it may happen that (p,r) influences an instruction I but the basic block B_I containing I is not in the specialization region of this triple because p does not dominate B_I. This problem can be remedied by duplicating code so as to make p dominate B_I. The second is that, as given, this definition does not take into account the size of a specialization region relative to the benefits obtained from its specialization. It may happen that an instruction I in a block B_I that is very far away from the point p is influenced by the value of a register r at p. If we include B_I in the specialization region, it is necessary to also include all of the blocks between p and B_I, even though

these blocks may not benefit from specialization. This could, in extreme cases, give rise to large specialization regions in order to include distant influenced instructions. This can be handled using a notion of *density* of influenced instructions, analogous to the notion of density of case labels used for code generation for `switch` statements [9, 44], to limit the specialization regions to code that contains a sufficiently high proportion of instructions that would benefit from the specialization. Currently, `Alto` neither duplicates code nor does it take the density into account.

5.2.4.3 Transforming the Program

The code transformations that are effected during specialization can be quite involved. They can depend on the type of instruction being specialized, the operand being specialized for, and the particular value of the operand. They can cause nontrivial restructuring of the control flow graph of the program, e.g., when the outcome of a conditional branch can be determined. Because of the involved nature of these transformations, and since this functionality is already available elsewhere in our system in the routines that implement constant propagation and constant folding (cf. Section 4.2.1), we do not have any separate code to implement all these transformations specifically for value-profile-based specialization. Instead, when specializing for a triple (p, v, r), we simply create a clone of the specialization region for that triple and insert a test at program point p that tests r and transfers control to the clone if r's value is not v (cf. Figure 5.2). No further work specific to value-profile-based specialization is necessary beyond this: the actual specialization of the code then takes place in the course of normal constant propagation and constant folding/strength reduction.

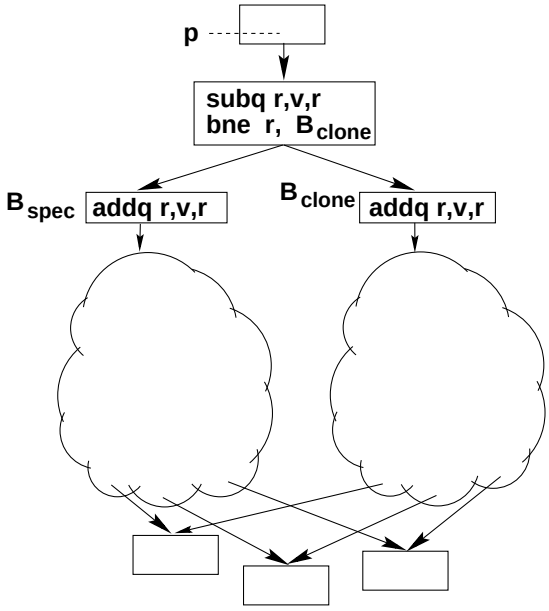


Figure 5.2: Specialization transformation

Given a specialization triple (p, r, v), a variety of idioms may be used to implement the test inserted at the program point *p*, depending on the magnitude of the value *v* and whether or not there is a free register available. If a free register r' is available, we simply compute the difference of *r* and *v* into r', then conditionally branch to the cloned code if r' is not zero. If there are no free registers available, if *v* is small enough to be an immediate operand the following pair of instructions is inserted (cf. Figure 5.2): [3].

```
subq r, v, r     # r := r - v
bne r, B_clone   # if (r ≠ 0) goto B_clone;
                 # else fall through to B_spec
```

To compensate for the effect of the `subq` instruction, we add the instruction '`addq r, v, r`' at the entry to both the original specialization region and its clone.

[3]The meaning of the Alpha machine instructions is explained in Appendix A

The actual specialization subsequently takes place during constant propagation and constant folding (cf. Section 4.2.1). Note, that constant propagation is able to derive information from conditionals [76]. If a basic block ends with a '`beq r, ...`' instruction ("branch if r is 0"), then the successor of this block on the true-branch will receive the information that r contains 0, while that on the false-branch will receive the information that r is non-zero; `bne` instructions ("branch if not equal to 0") are treated analogously. This turns out to be crucial for carrying out value specialization.

For the transformation sequence described above, constant propagation determines, from the instruction '`bne r,` B_{clone}' inserted as discussed above, that the register `r` has the value 0 at entry to the successor along the false-branch, i.e., the block B_{spec}. From this, it determines that after the instruction '`addq r, v, r`' in B_{spec}, the register `r` has the value v. This information is then propagated through the code fragment being specialized, and is used to carry out various optimizations as discussed above.

As an example of the effectiveness of our approach consider the code in Figure 5.3, which is part of the function `killtime()` in the SPEC95 benchmark `m88ksim`. The left hand side shows the unspecialized code. The code on the right hand side has been specialized for $r16 = 1$. The number of instructions is significantly reduced. Note, that the code shown represents a loop and that the test whether $r16 = 1$ hold occurs outside of this loop.

```
loop: cmpult   r6, r16, r0
      lda      r1, 0(r16)
      cmovne   r0, r6, r1
      ldl      r2, 4(r4)
      subl     r6, r1 ,r0
      stl      r0, 0(r4)
      cmpult   r2, r16, r0
      lda      r1, 0(r16)
      cmovne   r0, r2, r1
      ldl      r5, 8(r4)
      subl     r2, r1 ,r0
      stl      r0, 4(r4)
      cmpult   r5, r16, r0
      lda      r2, 0(r16)
      cmovne   r0, r5 ,r2
      ldl      r1, 12(r4)
      subl     r5, r2 ,r0
      stl      r0, 8(r4)
      cmpult   r1, r16, r0
      lda      r25, 0(r16)
      cmovne   r0, r1, r25
      lda      r4, 16(r4)
      subl     r1, r25, r1
      cmpult   r4, r3 ,r0
      stl      r1, -4(r4)
      bne      r0, loop
```

(a) unspecialized

```
loop: cmpult   r31, r6 ,r0
      ldl      r1, 4(r4)
      subl     r6, r0 ,r0
      stl      r0, 0(r4)
      cmpult   r31, r1, r0
      ldl      r2, 8(r4)
      subl     r1, r0 ,r0
      stl      r0, 4(r4)
      cmpult   r31, r2 ,r0
      ldl      r1, 12(r4)
      subl     r2, r0 ,r0
      stl      r0, 8(r4)
      cmpult   r31, r1 ,r25
      lda      r4, 16(r4)
      subl     r1, r25,r1
      cmpult   r4, r3 ,r0
      stl      r1, -4(r4)
      bne      r0, loop
```

(b) specialized for *r*16=1

Figure 5.3: Effect of value specialization on a node in `m88ksim::killtime()`

We have also implemented predicate profiling (cf. Section 5.2.3) for resolving pointer aliasing relationships. If we can determine whether or not two pointers are aliases in a frequently executed fragment of code, we can use this information in a variety of optimizations, including the avoidance of redundant memory accesses, and in instruction scheduling. For example, in the SPEC95 benchmark `m88ksim`, predicate profiling allows us to determine that three pointers (registers $r2$,$r17$,and $r18$) in a heavily executed loop within the function `alignd()` are usually not aliased; this information is used to eliminate several redundant memory accesses and thereby effect a significant speed improvement. Figure 5.4 shows the unspecialized code. The specialized version of the code is shown in Figure 5.5. The number of instructions has been reduced by one third. Note, that the code shown represents a loop and that the test whether the predicate is true occurs outside of this loop.

```
loop:
  ldl    r3, 0(r18)
  subl   r27, 4, r27
  ldl    r0, 0(r2)
  and    r3, 1, r3
  bis    r0, r3 ,r0
  stl    r0, 0(r2)
  ldl    r0, 0(r18)
  zapnot r0, 15, r0
  srl    r0, 1, r3
  stl    r3, 0(r18)
  ldl    r0, 0(r17)
  sll    r0, 31, r0
  bis    r3, r0 ,r0
  stl    r0, 0(r18)
  ldl    r0, 0(r17)
  zapnot r0, 15, r0
  srl    r0, 1, r0
  stl    r0, 0(r17)
  ldl    r0, 0(r18)
  ldl    r3, 0(r2)
  and    r0, 1, r0
  bis    r3, r0 ,r0
  stl    r0, 0(r2)
  ldl    r0, 0(r18)
  zapnot r0, 15, r0
  srl    r0, 1, r3
  stl    r3, 0(r18)
  ldl    r0, 0(r17)
  zapnot r0, 15, r0
  srl    r0, 1, r0
  stl    r0, 0(r17)
  ldl    r0, 0(r18)
  lda    r4, -4(r27)
  ldl    r3, 0(r2)
  and    r0, 1, r0
  bis    r3, r0 ,r0
  stl    r0, 0(r2)
  ldl    r0, 0(r18)
  zapnot r0, 15, r0
  srl    r0, 1, r0
  stl    r0, 0(r18)
  ldl    r3, 0(r17)
  sll    r3, 31, r3
  bis    r0, r3 ,r0
  stl    r0, 0(r18)
  ldl    r0, 0(r17)
  zapnot r0, 15, r0
  srl    r0, 1, r0
  stl    r0, 0(r17)
  ldl    r0, 0(r18)
  ldl    r3, 0(r2)
  and    r0, 1, r0
  bis    r3, r0 ,r0
  stl    r0, 0(r2)
  ldl    r0, 0(r18)
  zapnot r0, 15, r0
  srl    r0, 1, r3
  stl    r3, 0(r18)
  ldl    r0, 0(r17)
  sll    r0, 31, r0
  bis    r3, r0 ,r0
  stl    r0, 0(r18)
  ldl    r0, 0(r17)
  zapnot r0, 15, r0
  srl    r0, 1, r0
  stl    r0, 0(r17)
  bge    r4, loop
```

Figure 5.4: Unspecialized code fragment from `m88ksim::align()`

```
loop:
  ldl    r4, 0(r18)
  subl   r27, 4, r27
  ldl    r0, 0(r2)
  and    r4,1,r3
  bis    r0,r3 ,r7
  zapnot r4,15,r0
  srl    r0,1,r4
  ldl    r0,0(r17)
  sll    r0,31,r3
  bis    r4,r3 ,r6
  zapnot r0,15,r0
  srl    r0,1,r5
  addl   r7,r31,r3
  and    r6,1,r0
  bis    r3,r0 ,r4
  zapnot r6,15,r0
  srl    r0,1,r3
  addl   r5,r31,r0
  sll    r0,31,r0
  bis    r3,r0 ,r6
  zapnot r5,15,r0
  srl    r0,1,r0
  addl   r0,r31,r5
  lda    r7,-4(r27)
  addl   r4,r31,r3
  and    r6,1,r0
  bis    r3,r0 ,r4
  zapnot r6,15,r0
  srl    r0,1,r0
  sll    r5,31,r3
  bis    r0,r3 ,r0
  zapnot r5,15,r3
  srl    r3,1,r5
  addl   r4,r31,r4
  and    r0,1,r3
  bis    r4,r3 ,r3
  stl    r3,0(r2)
  zapnot r0,15,r0
  srl    r0,1,r3
  addl   r5,r31,r0
  sll    r0,31,r0
  bis    r3,r0 ,r0
  stl    r0,0(r18)
  zapnot r5,15,r0
  srl    r0,1,r0
  stl    r0,0(r17)
  bge    r7,loop
```

Figure 5.5: Specialized code fragment from `m88ksim::align()`

5.3 Experimental Setup

For the experimental evaluation we used the 18 programs from the SPEC95 benchmark suite [65]. The programs were compiled with the DEC C (FORTRAN) compiler V5.2-036 (V3.8-064) invoked as `cc -O4` (`f77 -O4`), with additional linker options (`-r -d -z -non_shared`) to retain relocation information and produce statically linked executables.[4] Both the initial execution frequency profiles as well as the value profiles for each program were obtained using the SPEC95 training inputs; the execution times reported were then obtained using the SPEC95 reference inputs.

The timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 processor with a split primary cache (8 kB each of instruction and data cache), 96 kB of on-chip secondary cache, 2 MB of off-chip backup cache, and 512 MB of main memory, running Digital Unix/Alpha V4.0B (Rev. 564). In each case, the execution time reported is the smallest time of 10 runs.

5.4 Experimental Results

Table 5.1 compares, for each benchmark, the total number of program points that could have been profiled/specialized (column 2) with the number that were actually profiled (column 3) and the number that were then specialized (column 4). This indicates that our computation of the cost/benefit tradeoffs is highly selective: the small number of points chosen for profiling keeps the value profiling overhead small, while the small number of points chosen for specialization keeps the code growth modest.

[4] We use statically linked executables because `Alto` relies on the presence of relocation information for its control flow analysis. The Digital Unix/Alpha linker `ld` refuses to retain relocation information for non-statically-linked executables.

Program	#Points	#Profiled	#Specialized
compress	15536	30	0
gcc	264020	2944	219
go	63895	348	4
ijpeg	46859	103	2
li	28168	54	5
m88ksim	36909	106	8
perl	78976	192	5
vortex	97037	55	6
applu	92323	171	7
apsi	108705	184	20
fpppp	90288	102	6
hydro2d	92276	139	6
mgrid	85826	10	1
su2cor	93673	111	5
swim	83660	26	0
tomcatv	82586	16	1
turb3d	90543	67	5
wave	109267	123	10

Table 5.1: Extent of profiling and specialization

Table 5.2 documents the code growth caused by value specialization. Column 2 states the code size of the benchmarks optimized with the `Alto` optimizer (cf. Chapter 4) without value specialization, column 3 states the size with value specialization. Column

4 has the ratio. The code growth is very moderate, especially for the integer subset of the benchmarks where it does not exceed 1%. For floating point subset it does not exceed 3.5%.

Program	Plain `Alto` (kB)	Specialized (kB)	Ratio
`compress`	74688	74688	1.000
`gcc`	1183488	1195584	1.010
`go`	296832	297920	1.004
`ijpeg`	213312	213376	1.000
`li`	125056	125120	1.001
`m88ksim`	174848	175296	1.003
`perl`	357248	357568	1.001
`vortex`	433216	433236	1.000
`applu`	427712	442624	1.035
`apsi`	494656	502208	1.015
`fpppp`	417728	418816	1.003
`hydro2d`	427264	437632	1.024
`mgrid`	399744	399780	1.000
`su2cor`	433600	442432	1.020
`swim`	388672	388672	1.000
`tomcatv`	387008	387024	1.000
`turb3d`	421056	422592	1.004
`wave`	498432	509632	1.022

Table 5.2: Code growth due to specialization

Program	Plain Alto (sec)	Specialized (sec)	Ratio
compress	259.8	259.8	1.000
gcc	232.9	229.8	0.987
go	304.0	299.3	0.985
ijpeg	328.1	327.9	1.000
li	254.6	248.8	0.977
m88ksim	224.2	194.3	0.867
perl	182.0	175.1	0.962
vortex	316.4	314.2	0.993
applu	357.6	354.4	0.991
apsi	194.5	189.6	0.975
fpppp	418.2	393.7	0.941
hydro2d	425.9	426.0	1.000
mgrid	339.9	328.6	0.967
su2cor	217.0	214.7	0.989
swim	265.0	264.9	1.000
tomcatv	283.1	279.2	0.986
turb3d	336.2	336.5	1.001
wave	223.7	217.9	0.974

Table 5.3: Execution time impact of value-profile-based specialization

The timing measurements are shown in Table 5.3. Column 2 states the execution time of the benchmarks optimized with the Alto optimizer without value specialization, column 3 states the execution time with value specialization. Column 4 has the ratio. It

can be seen from these numbers that automatic value-profile-based specialization can yield significant performance improvements for nontrivial programs. The programs that benefit the most are `li`, `m88ksim`, `perl`, `apsi`, `fpppp`, and `mgrid`.

The sources of performance improvements for these benchmarks are discussed below. There is, however, one caveat. In our system, value-profile-based specialization is carried out after subroutine inlining. Because of this, the code structure encountered during specialization, and the subroutines associated with the specialized code fragments, may not always correspond to those of the source program.

`li` : Sequences of independent conditionals in functions `xleval()` and `sweep()` are transformed so that the common case is tested first. A `switch` statement in the function `livecar()` is transformed so that the common case did not go through a jump table.

`m88ksim` : Predicate profiling allows us to determine that three pointers in the function `alignd()` are unaliased in the common case, allowing the elimination of several load and store instructions in that function. The function `killtime()` is specialized for an argument of 1.

`perl` : The function `memmove()` is specialized for the single byte move. The (internal) function `OtsDivide64Unsigned()`, which emulates integer division (since the Alpha does not have an integer division instruction), is specialized for the divisor 16.

`apsi` : Specialization allows several multiplication operations in the subroutine `pset`, and subroutines inlined into it, to be strength-reduced because one of the operands is zero.

`fpppp` : The common case for a computed goto statement in the subroutine `efill` is specialized.

`mgrid` : Specialization allows a multiplication operation in subroutine `resid` to be strength-reduced because one of the operands is zero.

Calder *et al.* report significant benefits from specializing the `hydro2d` benchmark [12]. To our surprise, however, we were not able to find significant specialization opportunities in this program: an examination of the code suggests that this may be due to improvements in the DEC FORTRAN compiler since their work was carried out.

CHAPTER 6

CODE COMPRESSION

In recent years there has been an increasing trend towards the incorporation of computers into a wide variety of devices, such as palm-tops, telephones, embedded controllers, etc. In many of these devices, the amount of memory available is limited, due to considerations such as space, weight, power consumption, or price. At the same time, there is an increasing desire to use more and more sophisticated software in such devices, such as encryption software in telephones, or speech or image processing software in laptops and palm-tops. Unfortunately, an application that requires more memory than is available on a particular device will not be able to run on that device. This makes it desirable to try and reduce the size of applications where possible. This chapter explores the use of object code modification to reduce code size and describes the implementation and experimental results of a code compressor based on `Alto`.

Our envisioned application scenario rules out certain compression schemes, described in the introduction of this dissertation. Compression that results in a program representation that needs to be decompressed before execution is undesirable for two reasons. First, extra memory is necessary to decompress the program. Second, the time overhead for decompression may be prohibitively big. Also undesirable are interpretive schemes because they will slow down execution and require the introduction of some form of runtime system and possibly changes to the operating system. Hence we make a tradeoff between compression ratio and execution speed/system complexity. [1]

[1] Using profiling information we could take this even further by excluding hot spots in the program from compression. However, we have not implemented this in our system yet.

Reductions in code size in our system come from two sources: aggressive (interprocedural) application of what are essentially classical compiler analyses and optimizations; and *code factoring*, a term we use to refer to a variety of techniques to identify and "factor out" repeated instruction sequences. Even though our compression techniques are applied to object code, they can be quite easily incorporated into compilers capable of interprocedural code transformations.

The overall structure of the code compressor is depicted in Figure 6.1. It is very similar to the optimizer described in Chapter 4.

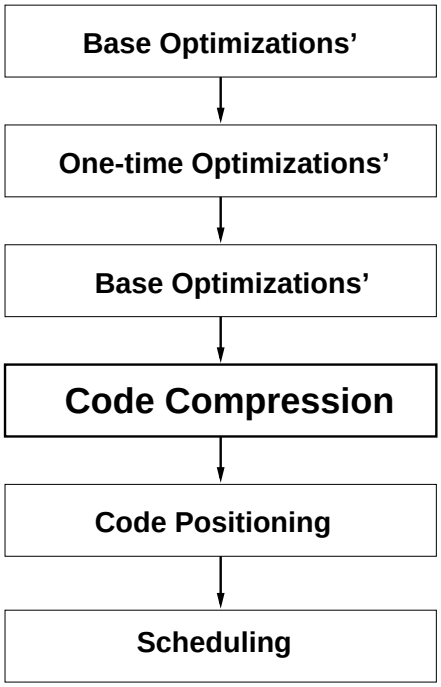


Figure 6.1: Phases of the code compressor based on `Alto`

Base Optimizations and **One-time Optimizations** have been changed so that optimizations that may increase code size are not invoked, e.g., there will be no inlining except for functions that have only a single call site. A few factoring transformations

have been added to the **Base Optimizations** to reduce code size. The new phase **Code Compression** contains the gist of our compression transformations.

The following sections describe the most relevant factoring transformations performed by the code compressor, and evaluates their effectiveness on the SPECint95 benchmark suite.

Related Work: Much of the earlier work on code compression, aiming at yielding smaller executables treated an executable program as a simple linear sequence of instructions. Early work by Fraser *et al.* used a suffix tree construction to identify repeated instruction sequences within such a linear sequence [35]. Such repeated sequences were then abstracted out into functions. Applied to a range of Unix/VAXutilities, this technique managed to reduce code size by a factor of about 7% on the average. A shortcoming of this approach is that since it relies on a purely textual interpretation of a program, it is sensitive to superficial differences between code fragments, e.g., due to differences in register names, that may not actually have any effect on the behavior of the code. This shortcoming was addressed by Baker, using parameterized suffix trees [6]; by Cooper and McIntosh, using register renaming [21] (Baker and Manber [7] discuss a similar approach); and by Zastre, using parameterized procedural abstractions [77]. The main idea is to rewrite instructions so that instead of using hard-coded register names, the (register) operands of an instruction are expressed, if possible, in terms of a previous reference (within the same basic block) to that register. Further, branch instructions are rewritten, where possible, in pc-relative form. These transformations allow the suffix tree construction to detect the repetition of similar but not lexically identical instruction sequences. Cooper and McIntosh obtain a code size reduction of about 5% on the average using these techniques on classically optimized code (in their implementation, classical optimizations achieve a code size reduction of about 18% compared to unoptimized code).

Any approach that treats a program as a simple linear sequence of instructions, as in

the suffix-tree-based approaches mentioned above, will suffer from the disadvantage of having to work with a particular ordering of instructions and basic blocks. There may be many reasons why two "equivalent" computations may map to different instruction sequences in two different parts of a program. The first, and most obvious, is that there may be differences in register usage and branch labels. Differences in the actual sequence of instructions produced may also arise due to instruction scheduling, or because of profile-directed code layout to improve instruction cache utilization [57].

Our approach to code compression will be somewhat different. Instead of treating a program as a simple linear sequence of instructions, we work with its (interprocedural) control flow graph. We use a scheme similar to [7] to identify "similar" basic blocks. If two blocks that are similar are found to not be identical, we try to rename registers—using a technique somewhat different from that of Cooper and McIntosh—in an attempt to make them identical. We use the notions of dominators and post-dominators to detect identical subgraphs of the control flow graph, larger than a single basic block, and that can be abstracted out into a procedure. Finally, we identify and take advantage of architecture-specific code idioms, e.g., for saving and restoring specific sets of registers at the entry to and return from functions.

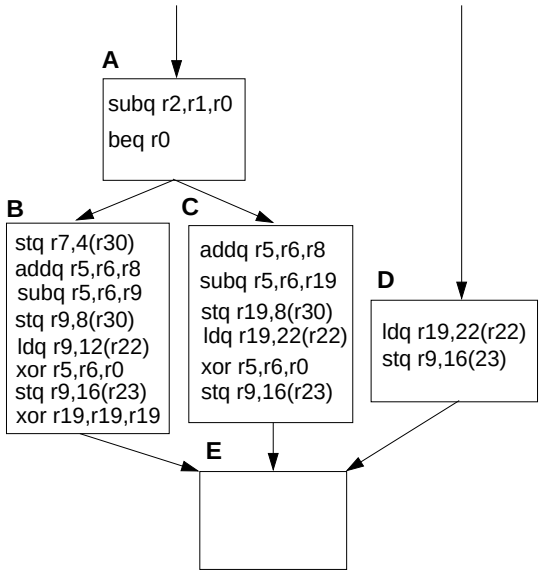
By showing how "equivalent" code fragments can be detected and factored out without having to resort to purely linear treatments of code sequences *as in suffix-tree-based approaches), our main contribution is to set up a framework for code compression that can be more flexible in its treatment of which code fragments are considered "equivalent." For example, while our current implementation searches for sets of basic blocks that contain identical instruction sequences, it is straightforward to generalize this component of the system to consider use-def chains (cf. Section 3.2), and thereby handle differences in the sequence of instructions arising out of instruction scheduling decisions. Similarly, the treatment of single-entry single-exit regions in Section 6.3.2 focuses on

structural properties of control flow graphs rather than any particular linearization: this allows it to handle differences in code sequences arising out of profile-directed code layout. We believe that the added flexibility gained from our approach can be useful in improving the results of code compression. A secondary contribution is to show that significant reductions in code size can be obtained without having to resort to extraneous structures such as suffix trees, by using information already available, e.g., the control flow graph and dominator/postdominator trees.

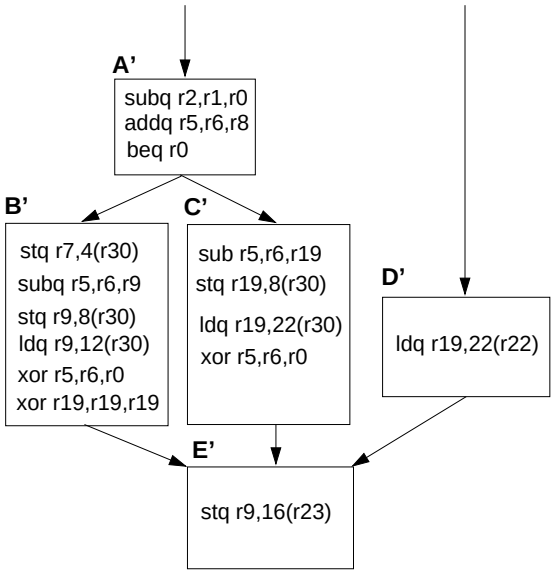
6.1 Local Factoring

The local factoring transformation was inspired by an idea of Knoop *et al.* [45]. It tries to merge identical code fragments by moving them to a point that pre- or post-dominates all the occurrences of the fragments. We have implemented a local variant of this scheme which we describe using the example depicted in Figure 6.2. [2].

[2]The meaning of the Alpha machine instructions is explained in Appendix A



(a) before



(b) after

Figure 6.2: Local factoring

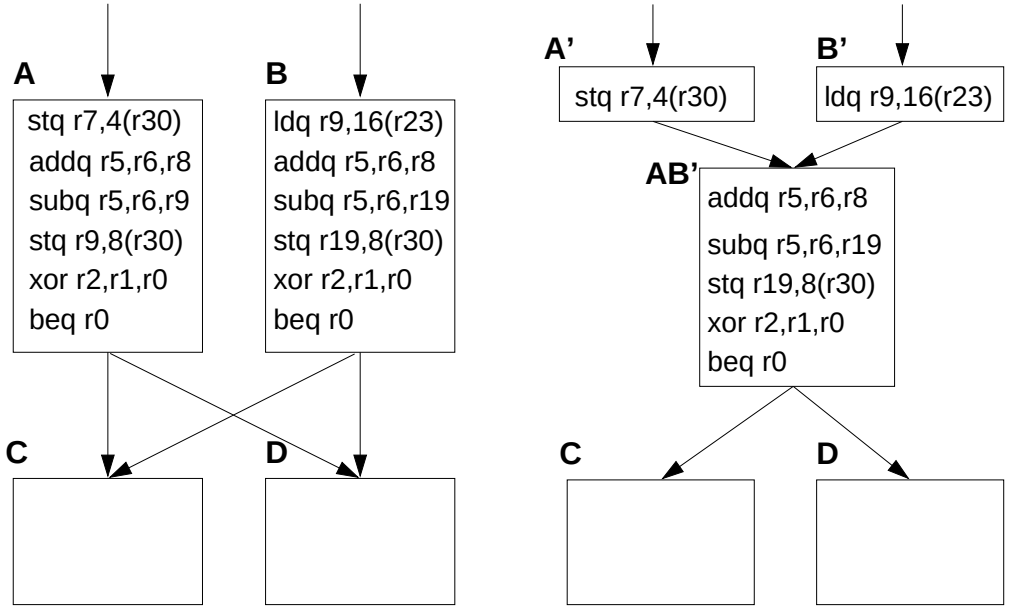
The left hand side of the figure shows an assembly code flowchart with a conditional branch (`beq r0`) in block A. Blocks B and C contain the same instruction '`addq r5,r6,r8`.' Since these instructions do not have backward dependencies with any other instruction in B or C, we can safely move them into block A just before the `beq` instruction, as shown in the right hand side of Figure 6.2. Similarly, blocks B, C, and D share the same store instruction '`stq r9,16(r23)`,' and since these instructions do not have forward dependencies with any other instruction in B, C,and D, they can be safely moved into block E. In this case it is not possible to move the store instruction from B and C into A because, due to the lack of aliasing information, there are backward dependencies to the load instructions (`ldq`) in B and C. In general, however, it might be possible to move an instruction either up or down. In this case we prefer to move it down since moving it up will eliminate exactly one copy while moving it down might eliminate several copies.

Our scheme uses register reallocation to make this transformation more effective. For example, the `subq` instructions in B and C write to different registers ($r9$ and $r19$). We can, however, rename $r9$ to $r19$ in B, thereby making the instructions identical. Another opportunity rests with the `xor` instructions in B and C. Even though they are identical we can not move them into A because they write register $r0$ which is used by the conditional branch. Reallocating $r0$ in A to another register which is dead at the end of A will make the transformation possible. Dead registers can be conveniently found using the interprocedural register liveness analysis (cf. Section 3.1).

Local factoring is invoked together with the base optimizations. It will not move instructions that change the flow of control nor will it create new basic blocks. But unlike factoring schemes described subsequently it might change both the register allocation and the instruction schedule.

6.2 Intraprocedural Tail Merging or Cross Jumping

Tail merging is a classical compiler transformation [55]. We describe it using the example depicted in Figure 6.3.



(a) before (b) after

Figure 6.3: Cross jumping

We first look for basic blocks like A and B with a common tail of instructions and which branch to the same basic block(s). This can be done efficiently by going backwards and searching the predecessors of a basic block for common tails. A new basic block AB' is then created containing the common instruction sequences from A and B which are eliminated from their original locations. The shrunken basic blocks A' and B' will branch to AB'.

Tail Merging is invoked together with the base optimizations. It will perform register

renaming to make tails more similar but does not reschedule them. It only merges tails of basic blocks within the same function.

6.3 Procedural Abstraction

Procedural abstraction is the inverse operation of inlining (cf. Section 4.4.1). Given a single-entry single-exit code fragment C, procedural abstraction of C involves (i) creating a procedure f_C whose body is a copy of C; and (ii) replacing the appropriate occurrences of C in the program text by a function call to f_C. While the first step is not very difficult, at the level of assembly or machine code the second step involves a little work. Procedural abstraction can in principle be done by a compiler but often the intermediate representations used in the compiler do not provide enough support for this kind of transformation. What is needed is the possibility to invoke a subroutine while maintaining the environment (stackframe and register contents) of the caller. At the object code level such a subroutine invocation mechanism is usually provide by some sort of "jump-and-link" instruction, that transfers control to the callee and at the same time puts the return address into a register, but leaves the stackframe and the other registers untouched. Liveness analysis (cf. Section 3.1) will usually provide us with several possible scratch registers to hold the return address. Which one do we choose? A simple method is to calculate, for each register r, the number of occurrences of code fragment C that could use r as a return register. A register with the highest such figure of merit is chosen as the return register for f_C. If a single instance of f_C, using a particular return register, is not enough to abstract out all of the occurrences of C in the program, we may create multiple instances of f_C that use different return registers. We use a more complicated scheme when abstracting function prologs (cf. Section 6.3.3.1) and regions of multiple basic blocks (cf. Section 6.3.2).

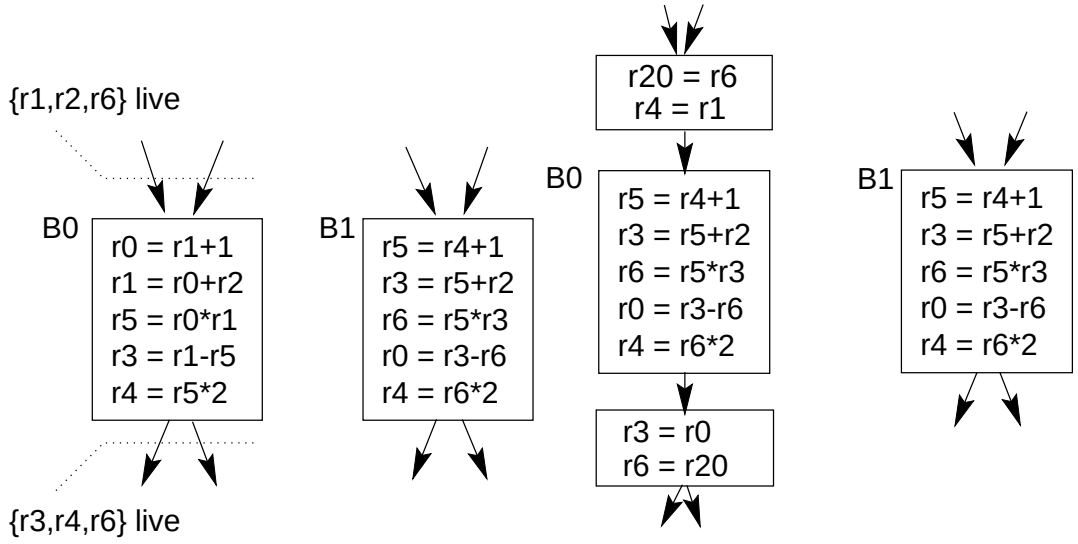
6.3.1 Procedural Abstraction for Individual Basic Blocks

Even though a basic block is a special case of single-entry single-exit region, it is handled separately because we do not require basic blocks to be identical in order to abstract them. We merely require that they be similar, viz. identical up to register allocation.

In order to determine whether two basic blocks are similar we adapt a technique from [6] and replace each mention of a register inside a basic block by the distance (measured in number of instructions) to its definition. These modified basic blocks are then sorted, yielding a partition of similar basic blocks.[3]

Next we examine each set of the partition in turn and attempt to convert similar basic blocks into identical basic blocks. The basic idea is very simple: registers are renamed "locally," i.e., within the basic block; and if necessary, register-to-register moves are inserted, in new basic blocks inserted immediately before and after the block being renamed, so as to preserve program behavior. An example of this is shown in Figure 6.4, where the we try to make the similar basic blocks B0 and B1 identical.

[3]We use a fairly canonical order: first considering the number instructions in the basic block, then the opcodes, and finally the (transformed) register names



(a) before (b) after

Figure 6.4: Example of basic-block-level register renaming

There are in general three reasons that keep us from simply copying a register allocation from one basic block to a similar basic block.

- Input registers. The two basic blocks might use registers that were not defined in the basic block and those might be different. This is exemplified by the use of *r*1 in B0 and *r*4 in B1, and can be compensated for by a introducing an additional move instruction '`r4 = r1`'.

- Output registers. The two basic blocks might define registers that are used outside of the basic block and those might be different. This is exemplified by the definition of *r*3 in B0 and *r*0 in B1, and can be compensated for by a introducing an additional move instruction '`r3 = r0`'. A subtle point is that we also need to make sure that the definition of *r*0 in B1 corresponding to the definition of *r*3 in

B0 is still available at the end of B1. If not the basic blocks are not really similar, and therefore we cannot make them identical.

- Live range conflicts. A register which is live through all of B0 might be defined in B1, thereby destroying a value needed after B0. We compensate for this by temporarily moving this register into another unused register. This is exemplified by register $r6$ which is temporarily moved into $r20$.

If none of these problems exists we can indeed copy the register allocation from one basic block literally to the other. If a problem exists, we have shown how to solve it by adding move instructions. We keep track of the number of move instructions necessary and will only add them if there is an overall benefit after abstraction. Note that the number of necessary move instructions required to make B0 identical to B1 might differ from the number of necessary move instructions to make B1 identical to B0, i.e., the relationship is asymetric. In order to cope with this we iterate over each set of similar basic blocks several times, trying to convert similar basic blocks into identical ones. In the first round we do not allow any move instruction to be added, in the next round we allow up to one move instruction to be add, then up to two and do on.

It also possible to employ some sort of meet in the middle approach to register renaming, where we do not try to make one basic block identical to another but where we just try to make them identical by changing both. We have not implemented this scheme yet.

A different approach to register renaming is described by Cooper and McIntosh [21]. They carry out register renaming at the level of entire live ranges: that is, when renaming a register $r0$ to a different register $r1$, the renaming is applied to an entire live range for $r0$. This has the advantage of not requiring additional register moves before and after a renamed block, as our approach does. However, it has the problem that register

renaming to allow the abstraction of a particular pair of basic blocks may interfere with the abstraction of a different pair of blocks.

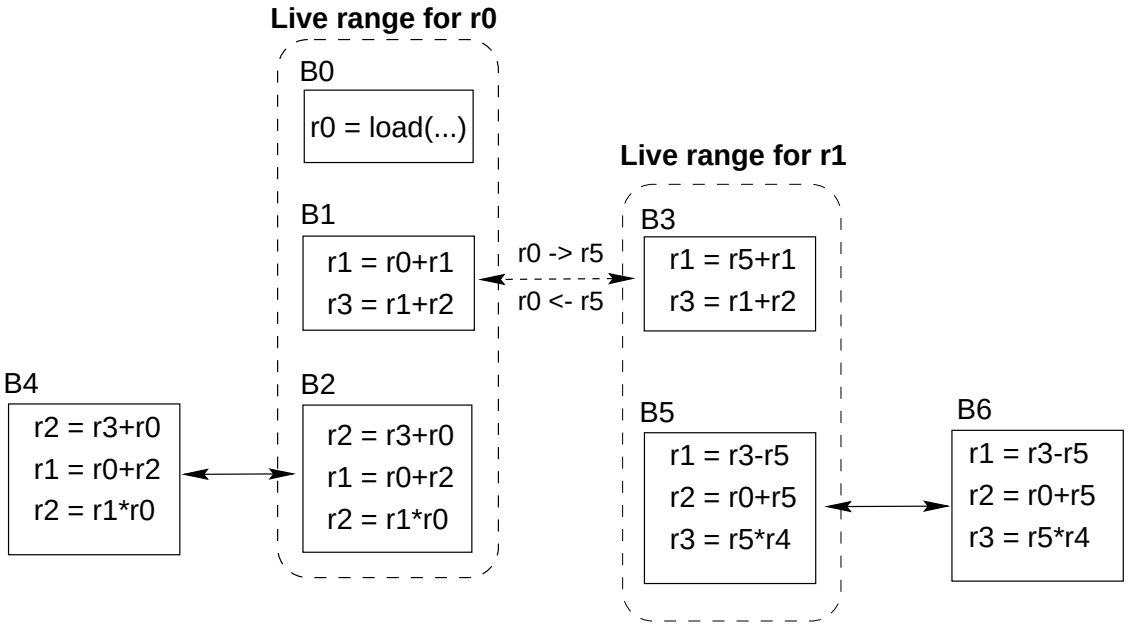


Figure 6.5: Interference effects in live-range-level register renaming

This is illustrated in Figure 6.5, where solid double arrows indicate identical basic blocks, while dashed double arrows indicate blocks that are not identical but which can be made identical via register renaming. Blocks B0, B1, and B2 comprise a live range for register *r*0, while B3 and B5 comprise a live range for *r*1. We can rename *r*0 to *r*5 in the live range for *r*0, so as to make blocks B1 and B3 identical, but this will cause blocks B2 and B4 to not be identical and therefore not abstractable into a function. We can also rename *r*5 to *r*0 in the live range for *r*1 so as to make it identical to B1, but this will interfere with the abstraction of blocks B5 and B6. Because of such interference effects, it is not clear whether live-range-level renaming produces results that are necessarily superior to basic-block-level renaming. Notice that the problem could be addressed by

judiciously splitting the live ranges: indeed, the local renaming we use can be seen as the limiting case of live-range-level renaming if splitting is applied until no live range spans more than one basic block.

Control Flow Separation

The approach described above will typically not be able to abstract two basic blocks that are identical except for an explicit control transfer instruction at the end. The reason for this is that if the control transfers are to different targets, the blocks will be considered to be different and so will not be abstracted. Moreover, if the control transfer instruction is a conditional branch, procedural abstraction becomes complicated by the fact that two possible return addresses have to be communicated.

To avoid such problems, basic blocks that end in an explicit control transfer instruction are split into two blocks: one block containing all the instructions in the block except for the control transfer, and another block that contains only the control transfer instruction. The first of this pair of blocks can then be subjected to renaming and/or procedural abstraction in the usual way.

6.3.2 Single-Entry/Single-Exit Regions

The discussion thus far has focused on the procedural abstraction of individual basic blocks. In general, however, we may be able to find multiple occurrences of a code fragment consisting of more than one basic block. We could, of course, factor out each basic block individually. But factoring out the entire region is more promising. In order to apply procedural abstraction to such a region R, at every occurrence of R in the program, we must be able to identify a single program point from which control enters R, and a single program point to which control returns after leaving R. It is not hard to see that any set of basic blocks R with a single entry point and a single exit point corresponds

to a pair of points (d, p) such that d dominates every block in R and p post-dominates every block in R; conversely, a pair of program points (d, p), where d dominates p and p post-dominates d, uniquely identifies a set of basic blocks with a single entry point and single exit point. Two such single-entry single-exit regions R and R' are considered to be identical if it is possible to set up a 1-1 correspondence $\simeq$ between their members such that $B_1 \simeq B_1'$ if and only if (i) B_1 is identical to B_1'; and (ii) if B_2 is a (immediate) successor of B_1 under some condition C, and B_2' is a (immediate) successor of B_1' under the same condition C, then $B_2 \simeq B_2'$. In order to determine whether two regions are identical we recursively traverse the two regions, starting at the entry node, and verifying that corresponding blocks are identical.

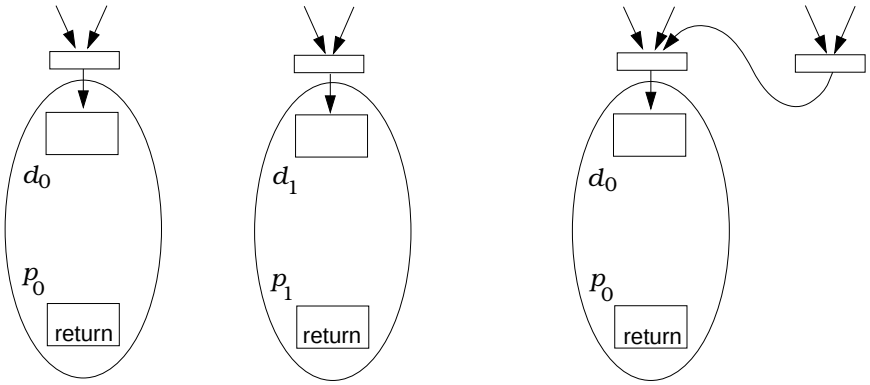
After procedural abstraction has been applied to individual basic blocks, we identify pairs of basic blocks (d, p) such that d dominates p and p post-dominates d. Each such pair defines a single-entry single-exit set of basic blocks. These sets of basic blocks are then partitioned into groups of identical regions, which then become candidates for further procedural abstraction.

To simplify the partition building process we compute a fingerprint for each region so that regions with different fingerprints will necessarily be different. These fingerprints are, 64-bit values: there are 8 bits for the number of basic blocks in the region and 8 bits for the total number of instructions, with the bit pattern `11...1` being used to represent values larger than 256; and the remaining 48 bits are used to encode the first (according to a particular preorder traversal of the region) 8 basic blocks in the region, with each block encoded using 6 bits: two bits give the type of the block, and four bits for the number of instructions in the block. The number of pairwise comparisons of fingerprints is reduced by distributing the regions over a hash table.

It turns out that applying procedural abstraction to a set of basic blocks is not as straightforward as for a single basic block, especially in a object code modifying imple-

mentation such as ours. The reason is that, in general, when the procedure corresponding to such a single-entry single-exit region is called, the return address will be put into a register whose value cannot be guaranteed to be preserved through that entire procedure, e.g., because the region may contain function calls. This means that the return address register has to be saved somewhere, e.g., on the stack. However, allocating an extra word on the stack, to hold the return address, can cause problems unless we are careful: allocating this space at the top of the stack frame can cause changes in the displacements of other variables in the stack frame, relative to the top-of-stack pointer; while allocating it at the bottom of the stack frame can change the displacements of any arguments that have been passed on the stack. If there is any address arithmetic involving the stack pointer, e.g., for address computations for local arrays, such computations may be affected by changes in displacements within the stack frame. These problems are somewhat easier to handle if the procedural abstraction is being carried out before code generation, e.g., at the level of abstract syntax trees [32]. At the level of assembly code [21, 35] or machine code (as in our work), it becomes considerably more complicated. There are, however, some simple cases where it is possible to avoid the complications associated with having to save and restore the return address when introducing procedural abstractions. Here, we identify two such situations.

In the first case, if we are given two identical regions (d_0, p_0) and (d_1, p_1), where p_0 and p_1 are return blocks (blocks from which control returns to the caller), there is no need to use procedural abstraction to create a separate function for these two regions. Instead, we can use an interprocedural version of the cross jumping transformation (cf. Section 6.2). The code in the region (d_1, p_1) is then simply replaced by a branch to d_0. The transformation is illustrated in Figure 6.6.



(a) before (b) after

Figure 6.6: Merging regions ending in `return`s via cross jumping

In the second case, given two identical regions (d_0, p_0) and (d_1, p_1) that we would like to abstract into a procedure, suppose that it is possible to find a register *r* that is (i) not live at entry to either of these regions; and (ii) whose value can be guaranteed to be preserved up to the end of the regions under consideration (*r* can be either a general-purpose register that is not defined within either region, or a callee-saved register that is already saved and restored by the functions in which the regions under consideration occur). In this case, when abstracting these regions into a procedure *p*, it is not necessary to add any code to explicitly save and restore the return address for *p*: the instruction to call *p* can simply put the return address in *r*, and the return instruction(s) within *p* can simply jump indirectly through *r* to return to the caller.

If neither of these conditions is satisfied, we determine whether the return address register can be safely saved in memory at entry to *p*, and restored at the end. For this, we use a conservative analysis to determine whether a function may have arguments passed on the stack, and which, if any, registers may be pointers into the stack frame. Given a

set of candidate regions to be abstracted into a representative procedure, we check the following:

1. For each function that contains a candidate region, it must be safe, with respect to the problems mentioned above, to allocate a word on the stack frame of the function.

2. There must be a register r_0 free at entry to each of the regions under consideration.

3. There must be a register r_1 free at the end of each of the regions under consideration.

4. There should not be any calls to `setjmp()`-like functions that can be affected by a change in the structure of the stack frame.

If these conditions are satisfied, *p* allocates an additional word on the stack on entry and saves the return address (passed via r_0) into this location; and loads the return address from this location (using r_1) and restores the stack frame on exit. The current implementation of the safety check described above is quite conservative in its treatment of function calls within a region. In principle, if we find that space can be allocated on the stack but have no free registers for the return address at entry or exit from the abstracted function, it should be possible to allocate an extra word on the stack in order to free up a register, but we have not implemented this yet.

6.3.3 Architecture-Specific Idioms

Apart from the general-purpose techniques described earlier for detecting and abstracting out repeated code fragments, there are machine-specific idioms that can be profitably exploited. In particular, the instructions to save and restore registers (the return address and callee-saved registers) in the prolog and epilog of each function generally have a

predictable structure because those registers are saved at predictable locations within the stack frame. For example, the standard calling convention for Digital Unix/Alpha treats register *ra* (or *r*26) as the return address register and registers *r*9 through *r*15 as callee-saved registers; these are saved at locations `0(sp)`, `8(sp)`, `16(sp)`, and so on where *sp* denotes the stack pointer register. Abstracting out such instructions can yield considerable savings in code size.

The register save/restore instructions in function prologs and epilogs typically follow the same sequence. In function epilogs this sequence of actions is reversed. However, the fact that different function prologs carry out a similar sequence of events does not imply that the same instruction sequences are encountered in the prologs of different functions: instruction scheduling can, and does, cause other instructions to be interspersed in between the code to save registers; a similar comment applies to function epilogs. Because of this, the techniques described earlier, which rely on identifying identical instruction sequences and/or basic blocks, may not always be able to factor out the instructions for saving/restoring registers in function prologs and epilogs. Instead, they are treated specially.

6.3.3.1 Abstracting Register Saves

In order to abstract out the register save instructions in the prolog of a function f into a separate function *g*, it is necessary to identify a register that can be used to hold the return address for the call from f to *g*. Liveness analysis is employed to find such a register. For each candidate register *r*, we first compute the savings that would be obtained if *r* were to be used for the return address for such calls. This is done by totaling up, for each function f where *r* is free at entry to f, the number of registers saved in f's prolog, i.e., the size of the prolog. We then choose a register *r* with maximum savings, and generate

a family of functions $Save^r_{r15}, \ldots, Save^r_{r9}, Save^r_{ra}$ that save the callee-saved registers and the return address register, and then return via register r. The idea is that function $Save^r_i$ saves register i and then falls through to function $Save^r_{i-1}$.

If a function has register r available before the prolog code, and subsequently saves registers $r9, \ldots, rk$, we can replace the prolog by a call to $Save^r_{rk}$.

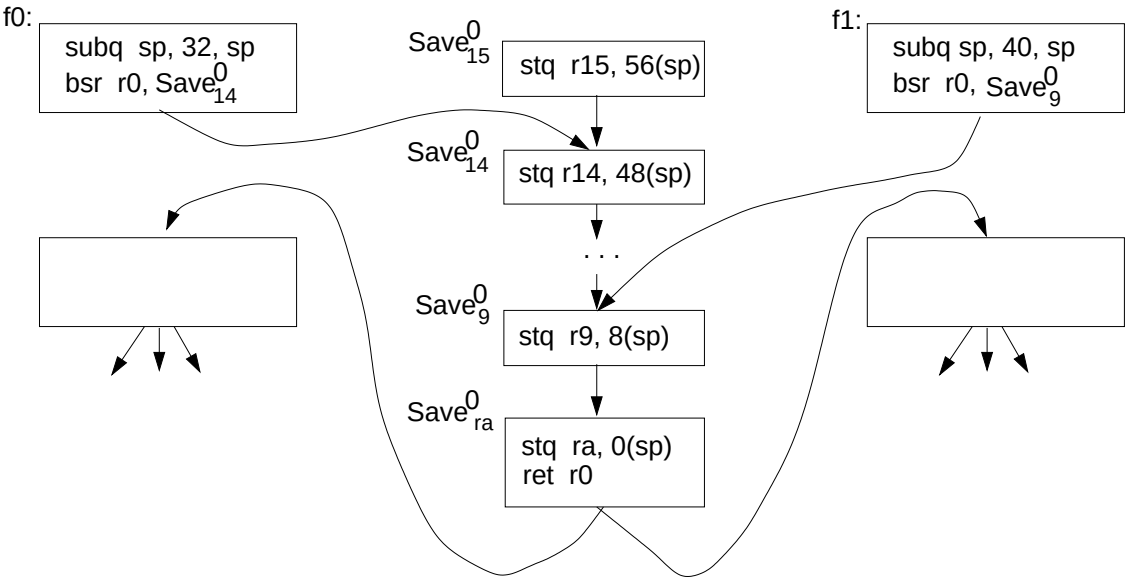


Figure 6.7: Example for function prolog factoring

As an example, suppose we have two functions `f0()` and `f1()`, such that `f0()` saves registers $r9, \ldots, r14$, and `f1()` saves only register $r9$. Assume that register $r0$ is free at entry to both these functions and is chosen as the return address register. The code resulting from the transformation described above is shown in Figure 6.7.

It may turn out that the set of functions subjected to this transformation do not use all of the callee-saved registers. For example, in Figure 6.7, suppose that none of the functions using return address register $r0$ save register $r15$. In this case, the code for the

function $Save^{0}_{r15}$ becomes unreachable and is subsequently eliminated.

A particular choice of return address register, as described above, may not account for all of the functions in a program. The process is therefore repeated, using other choices of return address registers, until either no further benefit can be obtained, or all functions are accounted for.

6.3.3.2 Abstracting Register Restores

The code for abstracting out register restore sequences in function epilogs is conceptually analogous to that described above, but with a few differences. If we were to simply do the opposite of what was done for register saves in function prologs, the code resulting from procedural abstraction at each return block for a function might have the following structure, with three instructions to manage the control transfers and stack pointer update:

```
...
bsr r1, Restore     # call register restore function
addq sp,k,sp        # deallocate stack frame
ret ra              # return
```

If we could somehow move the instruction for deallocating the stack frame into the function that restores saved registers, there would be no need to return to the function f whose epilog we are abstracting: control could return directly to f's caller (in effect realizing tail call optimization). The problem is that the code to restore saved registers is used by many different functions, which in general have stack frames of different sizes, and hence need to adjust the stack pointer by different amounts. The solution to this problem is to pass, as an argument to the function that restores registers, the amount by which the stack pointer must be adjusted. Since the return address register *ra* is guaranteed to be free at this point—it is about to be overwritten with f's return address prior to returning

control to *f*'s caller—it can be used to pass this argument.[4] Since there is now no need for control to return to *f* after the registers have been restored—it can return directly to *f*'s caller—we can simply jump from function *f* to the function that restores registers, instead of using a function call. The resulting code requires two instructions instead of three in each function return block:

```
...
move k,ra          # sp needs to be adjusted by k
br Restore         # jump to register restore function
```

The code in the function that restores registers is pretty much what one would expect. Unlike the situation for register save sequences discussed in Section 6.3.3.1, we need only one function for restoring registers. The reason for this is that there is no need to call this function: control can jump into it directly, as discussed above.

[note that this is essentially cross jumping but interprocedurally and with a parameter - the stack size]. This means that we do not have to generate different versions of the function with different return address registers. The overall structure of the code is analogous to that for saving registers: there is a chain of basic blocks, each of which restores a callee-saved register, with control falling through into the next block, which saves the next (lower-numbered) callee-saved register, and so on. The last member of this chain adjusts the stack pointer appropriately, loads the return address into a register, and returns. There is, however, one minor twist at the end. The amount by which the stack pointer must be adjusted is passed in register *ra*, so this register cannot be overwritten until after it has been used to adjust the stack pointer. On the other hand, since the memory location from which *f*'s memory address is to be restored is in *f*'s stack frame,

[4] In practice not all functions can be guaranteed to follow the standard calling convention, so it is necessary to verify that register *ra* is, in fact, being used as the return address register by *f*.

we can't adjust the stack pointer until after the return address has been loaded into *ra*. We get around this problem using the following instruction sequence:

```
...
addq sp, ra, sp   # sp := sp + ra ≡ new sp
subq sp, ra, ra   # ra := sp - ra ≡ old sp
ldq ra, 0(ra)     # ra := return address
ret ra
```

The resulting code for restoring saved registers, for the functions considered in the example illustrated in Figure 6.7, is shown in Figure 6.8.

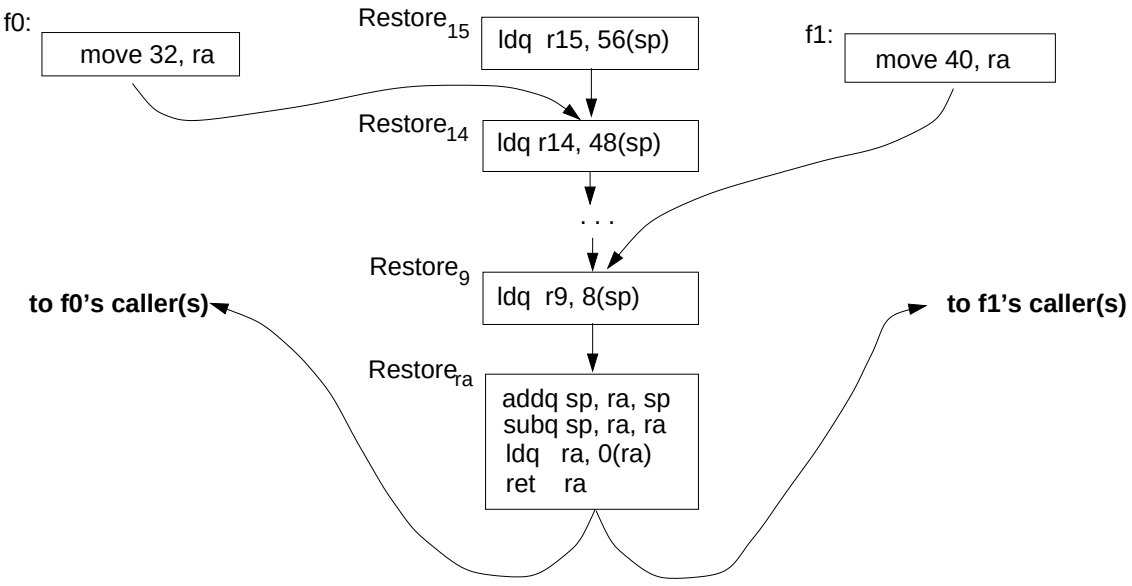


Figure 6.8: Example for function epilog factoring

We go through these contortions in order to minimize the number of registers used. If we could find another register that is free at the end of every function, we could load the return address into this register, resulting in somewhat simpler code. However, in

general it is not easy to find a register that is free at the end of every function. Moreover, since there is only one function that restores saved registers in the transformed code, the overall savings from this, even if we could find such a free register, would not be very significant. Compared to the obvious implementation described at the beginning of this subsection, the resulting code reduces the number of instructions necessary at each function return block from three to two, i.e., with a net savings of one instruction, at the cost of introducing three additional instructions into the function that abstracts the register restore instructions. It is therefore able to achieve a net savings, compared to the obvious implementation, if there are at least four functions in the program whose register restore actions can be abstracted as described above. The reason we go to such lengths to eliminate a single instruction from each return block is that there are a lot of return blocks, amounting to about 4%–8% of the basic blocks in a program (there is usually at least one—and, very often, more than one—such block for each function). The elimination of one instruction from each such block translates to a code size reduction of about 1%–2% overall (this may seem small, but to put it in perspective, consider that Cooper and McIntosh report an overall code size reduction of about 5% using suffix-tree based techniques [21]).

6.4 Experimental Setup

For the experimental evaluation we used the 8 programs from the SPECint95 benchmark suite [65]. The benchmarks were compiled with the DEC C compiler V5.2-036 invoked as `cc -O1`, which is the highest optimization level, that does not perform code size increasing transformations. Additional linker options (`-d -z -r -non_shared`) were needed to retain relocation information and produce statically linked executables.[5]

[5]We use statically linked executables because `Alto` relies on the presence of relocation information for the control flow graph construction. The Digital Unix/Alpha linker `ld` refuses to retain relocation

No execution frequency profiles were used. The execution times reported were generated using the benchmark reference inputs. The timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 (EV5) processor with a split primary cache (8 kB each of instruction and data cache), 96 kB of on-chip secondary cache, 2 MB of off-chip backup cache, and 512 MB of main memory, running Digital Unix/Alpha V4.0B (Rev. 564). In each case, the execution time reported is the smallest time of 10 runs.

6.5 Experimental Results

Table 6.1 compares, for each benchmark, the code size for the original (unoptimized) version (Column 2), the `Alto` optimized version using the regular optimizer without

Program	`cc` (kB) [norm]	`Alto` (kB) [n.]	`Alto'` (kB) [n.]	`Fac.` (kB) [n.]
`compress`	99 [1.000]	68 [0.691]	64 [0.651]	61 [0.612]
`gcc`	1362 [1.000]	1083 [0.796]	1032 [0.758]	972 [0.714]
`go`	341 [1.000]	278 [0.816]	266 [0.779]	254 [0.744]
`ijpeg`	262 [1.000]	186 [0.713]	178 [0.682]	166 [0.636]
`li`	179 [1.000]	115 [0.647]	110 [0.615]	101 [0.568]
`m88ksim`	228 [1.000]	162 [0.709]	152 [0.666]	142 [0.621]
`perl`	435 [1.000]	324 [0.745]	309 [0.710]	282 [0.650]
`vortex`	696 [1.000]	410 [0.589]	391 [0.562]	374 [0.538]
Geom. Mean	[1.000]	[0.710]	[0.674]	[0.632]

Table 6.1: Impact of code compression on code size

information for non-statically-linked executables.

profiles (Column 3), the `Alto` optimized version where code growing optimizations have been disabled (Column 4), and the smallest possible version using the factoring transformations (Column 5). Normalized numbers relative to Column 2 are also presented.

Using the factoring transformations we can reduce the code size by 36.8% in the average compared to the original. Of the this reduction 4.2% is due to factoring while the rest is due to optimizations performed by `Alto`.

Table 6.2 has the same structure as Table 6.1 but compares execution times instead of code sizes. As expected, the extra reduction in code size due factoring is obtained by a small penalty (7.5%) in execution time. However, on the average we are still 7.9% faster than the original executable.

Program	`cc` (sec) [norm]	`Alto` (sec) [n.]	`Alto`' (sec) [n.]	`Fac.` (sec) [n.]
`compress`	321.3 [1.000]	281.4 [0.876]	284.6 [0.886]	289.4 [0.901]
`gcc`	262.1 [1.000]	257.2 [0.981]	252.8 [0.964]	260.6 [0.994]
`go`	360.0 [1.000]	298.0 [0.828]	299.4 [0.832]	317.5 [0.882]
`ijpeg`	327.1 [1.000]	324.1 [0.991]	329.3 [1.007]	330.1 [1.009]
`li`	312.0 [1.000]	265.0 [0.849]	260.2 [0.834]	314.7 [1.009]
`m88ksim`	400.3 [1.000]	272.2 [0.680]	287.2 [0.717]	289.8 [0.724]
`perl`	257.0 [1.000]	214.4 [0.834]	209.3 [0.814]	230.6 [0.897]
`vortex`	470.4 [1.000]	351.9 [0.748]	359.1 [0.763]	425.5 [0.904]
Geom. Mean	[1.000]	[0.843]	[0.847]	[0.910]

Table 6.2: Impact of code compression on execution time

CHAPTER 7

FUTURE WORK

The research described in this dissertation suggests several avenues for future work. The most obvious of these would explore extensions to optimizations and code compression transformations.

Especially in the area of *profile guided optimizations* very few advances have been made that have proven to be beneficial enough to be incorporated into a production quality compiler. We believe that this is mostly due to the already mentioned "impedance mismatch" between the easily available low level profiling information and the high level intermediate representation inside most compilers. This problem does not exist in `Alto` which should make it an ideal platform for further studies with profile guided optimizations.

A related issue is *resource guided optimizations*. Many compilers perform optimizations such as inlining and loop-unrolling without (or very little) regard for the available resources of the underlying machine. This can lead to unexpected and counter-productive results. For example, excessive inlining can increase the amount of code that is executed frequently (the working set) beyond the size of the instruction cache thereby increasing cache misses and degrading performance. The same problem arises with loop-unrolling. An infamous example is the SPEC95 benchmark `fpppp` which contains loop that has been manually unrolled. This unrolled loop which accounts for most of the cycles spent in the benchmark results in a basic block with over 8000 instructions — far exceeding the instruction cache size. This suggests to leave such potentially harmful transformations

to an optimizer, like `Alto`, that works on a very low level, where it is easier to estimate resource usage

There are several interesting enhancements to guarded code specialization:

- Instead of specializing for the value with the highest benefit at a certain (program point, register) pair. We could specialize for the *n* most beneficial values, possibly inserting a sequence of tests to dispatch for the actual value.

- Quite often several registers might have a very skewed value distribution at a program point or the conditional distribution of values might be skewed. So instead of profiling for one register at time, we might want to simultenously profile several registers. This will require a scheme for computing combined benefits when the contents of several registers are known.

- We have already hinted that our value based profiling/specializion is just a special case of the more general predicate based profiling/specializion. The automatic computation of useful predicates to profile for seems challenging but promising extension.

Our current implementation of code compression on the object code level does not cope with scheduling very well. Only prolog/epilog factoring and local factoring are effective in the presence of rescheduled instruction sequences. It would be nice to extend basic block factoring so that not just basic blocks which are identical up to register reallocation but also basic block which are identical up to instruction scheduling can be factored. This might be a difficult task, since we probably have to compute the dependence graph of the instructions in each basic block and then search for isomorphic graphs. It would also be nice to extend this to the sub-basic block level allowing for factoring of parts of basic blocks.

APPENDIX A

ALPHA MACHINE INSTRUCTIONS

Instruction	Effect
`ldq ra,n(rb)`	load the quadword (8 bytes) at address $n + rb$ into *ra*
`ldl ra,n(rb)`	load the longwordd (4 bytes) at address $n + rb$ into *ra*
`stq ra,n(rb)`	store the quadword in *ra* at the address $n + rb$
`stl ra,n(rb)`	store the longword in *ra* at the address $n + rb$
`lda ra,n(rb)`	compute the address (value) $n + rb$ into *ra*
`and ra,rb,rc`	compute the bitwise and of *ra* and *rb* into *rc*
`bis ra,rb,rc`	compute the bitwise or of *ra* and *rb* into *rc*
`xor ra,rb,rc`	compute the bitwise xor of *ra* and *rb* into *rc*
`slr ra,rb,rc`	shift *ra rb* bits to the right into *rc*
`sll ra,rb,rc`	shift *ra rb* bits to the left into *rc*
`move ra,rb`	move *ra* into *rb*
`addq ra,rb,rc`	compute the sum of the quadwords in *ra* and *rb* into *rc*
`addl ra,rb,rc`	compute the sum of the longwords in *ra* and *rb* into *rc*
`subq ra,rb,rc`	compute the difference of the quadwords in *ra* and *rb* into *rc*
`subl ra,rb,rc`	compute the difference of the longwords in *ra* and *rb* into *rc*
`mulq ra,rb,rc`	compute the product of the quadwords in *ra* and *rb* into *rc*

Instruction	Effect
`br label`	branch unconditionally to label
`jmp (ra)`	branch unconditionally to the address in *ra*
`ret ra`	return to the address in *ra*
`bsr ra,label`	call the subroutine at label, storing the return address into *ra*
`jsr ra,(rb)`	call the subroutine at the address in *rb*, storing the return address into *ra*
`beq ra,label`	branch to label if $ra = 0$
`bne ra,label`	branch to label if $ra \neq 0$
`bge ra,label`	branch to label if $ra \geq 0$
`cmpeq ra,rb,rc`	set *rc* to 1 if $ra = rb$, 0 otherwise
`cmpne ra,rb,rc`	set *rc* to 1 if $ra \neq rb$, 0 otherwise
`cmpult ra,rb,rc`	set *rc* to 1 if $ra < rb$, 0 otherwise

Register	Synonym	Usage
*r*0	*v*0	subroutine result
$r1 - r25$		general purpose
*r*26	*ra*	return address for subroutine call
*r*27	*pv*	subroutine address for subroutine call
*r*28		general purpose
*r*29	*gp*	pointer into constant pool (global pointer)
*r*30	*sp*	stack pointer
*r*31	*zero*	hard wired to zero

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, Pa., May 1996.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, 8–12 July 1996. Springer.

[4] 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. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[5] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 149–159, May 1996.

[6] Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 71–80, San Diego, California, 16–18 May 1993.

[7] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 179–190, Berkeley, USA, June 15–19 1998. USENIX Association.

[8] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[9] Robert L. Bernstein. Producing good code for the `case` statement. *Software Practice and Experience*, 15(10):1021–1024, October 1985. See correction [44].

[10] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[11] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, Los Alamitos, December1–3 1997. IEEE Computer Society.

[12] B. Calder, P. Feller, and A. Eustace. Value profiling. *The Journal of Instruction-Level Parallelism*, 1, March 1999 (http://www.jilp.org/vol1).

[13] Paul Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[14] 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.

[15] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), March/April 1998.

[16] F. C. Chow. Minimizing register usage penalty at procedure calls. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–94, Atlanta, GE, USA, June 1988. ACM Press.

[17] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for Alpha/NT executables. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 17–23, Berkeley, CA, USA, August 1997. USENIX.

[18] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 80–89, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[19] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996.

[20] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: a case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[21] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. *ACM SIGPLAN Notices*, 34(5):139–149, May 1999.

[22] Digital Equipment Corporation. *Assembly Language Programmer's Guide*. Maynard, Massachusetts, 1994.

[23] R. Covington, S. Madala, V. Mehta, J. Jump, and J. Sinclair. Efficient simulation of parallel computer systems. *International journal in computer simulation*, 1,1:31–58, 1991.

[24] Dan Crove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 90–99, New York, NY, USA, June 1993. ACM Press.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[26] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Trans. on Softw. Eng.*, 18(2):89, February 1992.

[27] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. Technical Report TR99-07, The Department of Computer Science, University of Arizona, Friday, April 23 1999.

[28] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, San Diego, California, January 1998.

[29] Dawson R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, New York, May 1996. ACM Press.

[30] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, pages 131–144, St. Petersburg Beach, Florida, January 1996.

[31] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 358–365, New York, June 1997. ACM Press.

[32] Michael Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 263–276. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[33] Michael Franz and Thomas Kistler. Slim binaries. Technical Report ICS-TR-96-24, University of California,Irvine, June 1996.

[34] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[35] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

[36] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 122–133, New York, June 1997. ACM Press.

[37] David W. Goodwin. Personal communication, 1997.

[38] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. `gprof`: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[39] Rand Gray and Deepak Mulchandani. Object file formats. *Dr. Dobb's Journal of Software Tools*, 22(5), May 1997.

[40] Reed Hastings and Bob Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136. USENIX Association, 1992.

[41] Urs Hölzle and Ole Agesen. Dynamic versus static optimization techniques for object-oriented languages. *Theory and Practice of Object Sytems*, 1(3):167–188, 1995.

[42] Hans-Dieter Jankowski, Dietmar Rabich, and Julian F. Reschke. *ATARI-ST-Profibuch (in german)*. SYBEX, 1991.

[43] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[44] Sampath Kannan and Todd A. Proebsting. Short communication: Correction to "Producing Good Code for the case Statement", SPE 15: 1021–1024 (October 1985). *Software Practice and Experience*, 24(2):233–233, February 1994.

[45] 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.

[46] William Landi and Barbara Ryder. Pointer-induced aliasing: A problem taxonomy. In *ACM '91 Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, USA, January 1991.

[47] James R. Larus and Thomas J. Ball. Rewriting executable files to measure program behavior. Technical Report 1083, Computer Sciences Department, University of Wisconsin-Madison, March 1992.

[48] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.

[49] D. Lee, P. Crowley, J. Baer, T. Anderson, and B. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 27–38, New York, June 27–July 1 1998. ACM Press.

[50] Peter Lee and Mark Leone. Optimizing ML with run-time code modifcation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 137–148, New York, May 1996. ACM Press.

[51] D. B. Loveman and R. A. Faneuf. Program optimization-theory and practice. *ACM SIGPLAN Notices*, 10(3):97–102, March 1975.

[52] Thomas J. Marlowe, Barbara Ryder, and Michael Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Rutgers University, July 1995.

[53] Scott McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 71–79, Toronto, Canada, June 1991.

[54] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

[55] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[56] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, January 26–28, 1981. ACM SIGACT-SIGPLAN, ACM Press.

[57] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, NY, USA, June 1990. ACM Press.

[58] The Massively Scalar Compiler Project. The iloc intermediate language. Website http://www.crpc.rice.edu/MSCP/iloc.html, Rice University, 1995.

[59] Press Release. Pklite. Website http://www.pklite.com/news/pr_951003a.html, PKWARE, 1995.

[60] 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 USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 1–7, Berkeley, CA, USA, August 1997. USENIX.

[61] M. Ronsse and K. De Bosschere. JiTI : Tracing memory references for data race detection. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, volume 12 of *Advances in Parallel Computing*, pages 327–334, Amsterdam, February 1998. Elsevier, North-Holland.

[62] M. A. Ronsse and L. J. Levrouw. On the implementation of a replay mechanism. *Lecture Notes in Computer Science*, 1123:70–80, 1996.

[63] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report RADC-TR-69-313, Volume II, Applied Data Research, Incorporated, September 1969.

[64] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[65] SPEC. Cpu95 benchmarks. Website http://www.spec.org/, Standard Performance Evaluation Corporation, 1995.

[66] Amitabh Srivastava and Alan Eustace. ATOM—A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, Florida, June 1994.

[67] 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.

[68] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, Orlando, Florida, June 1994.

[69] MIPS Computer Systems. Riscompiler and c programmer's guide. Technical report, MIPS Computer Systems, 1990.

[70] Alpha Migration Tools. Decmigrate. Website http://www.digital.com/amt/decmigrate/, COMPAQ, 1998.

[71] Alpha Migration Tools. Freeport Express. Website http://www.digital.com/amt/freeport/, COMPAQ, 1998.

[72] 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, 1993.

[73] Robert Wahbe, Steven Lucco, and Susan L. Graham. Adaptable binary programs. Technical Report CS-94-137, Carnegie Mellon University, School of Computer Science, April 1994.

[74] David W. Wall. Global register allocation at link time. Technical Report 86/3, Digital Equipment Corporation, Western Research Lab, October 1986.

[75] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.

[76] Mark N. Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[77] M. J. Zastre. Compacting object code via parameterized procedural abstraction. Master thesis, Department of Computer Science, University of Victoria, Canada, 1993.