A Case for Automatic Run-Time Code Optimization

A Thesis presented

by

Eric Jay Feigin

To

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 5, 1999

Abstract

This thesis describes the design of a software system capable of automatically performing code optimizations at run-time. Rather than attempting to run all optimizations statically, a compiler produces an executable capable of monitoring its own run-time behavior and performing on-the-fly optimizations which take advantage of current execution patterns. This technique, which we call *dynamic optimization*, potentially adds a high degree of adaptability to code optimization, postponing optimization decisions until the exact nature of the input set or program phase is known, and allowing these decisions to change as program behavior changes. The goal of this thesis is to demonstrate that dynamic optimization is both feasible and profitable.

Table of Contents

Chapter 1 Introduction

The goal of code optimization is to streamline code in ways either difficult or impossible for the programmer to accomplish. Programs are typically written in high-level languages, often with the intent of both generality and target-independence; code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine-specific features. Current trends towards portable languages like Java are widening the gap even further between programmers and the machines which execute their code; this makes code optimization even more important for obtaining maximal benefit from new microarchitectural features.

1.1 Static Code Optimization

The traditional approach to code optimization is the compile-time optimizer. Since the compiler already has the job of converting source code into machine code, it seems quite natural for it to also do code optimization. Compile-time code optimization is seen as a way for the compiler to streamline the execution of the program.

Unfortunately, there are obstacles which can limit the effectiveness of compile-time code optimization. One is of these is the dependence of compile-time optimizations on the somewhat arbitrary structure of program code. In particular, procedure boundaries inhibit the effectiveness of many optimizations. While studies have shown that there are significant benefits to be gained from optimizing across procedure boundaries [23,24], finding and exploiting interprocedural opportunities can be quite challenging.

Aggressive function inlining can remove many procedure boundaries entirely, but comes at the cost of increased code size, which, among its other drawbacks, can greatly increase cache misses. Consistently effective heuristics to determine when function inlining is worthwhile have yet to be demonstrated. Another strategy for identifying interprocedural optimization opportunities is to employ interprocedural dataflow analysis techniques. However, some evidence argues that such methods are too limited in their effectiveness to be worth the additional complexity they create in the compiler [24].

Additionally, some program code may be completely unavailable at compile-time. For example, it is common for the standard C libraries to be included in a program as dynamically linked libraries. Because these routines are not loaded in until run-time, calls to them represent a complete barrier to possible compile-time optimization.

Another difficulty for compile-time optimization is that very few effective code optimizations can be done "for free", i.e. guarantee increased program efficiency with no negative side effects. Most optimizations have drawbacks, limiting their potential effectiveness. Some, such as loop unrolling, increase code size. Others, such as hot-cold optimization, apply transformations which streamline a particular region of code, quite possibly at the expense of other regions [7]. Choosing which optimizations to apply and where to apply them becomes a matter of effectively managing a complex system of tradeoffs.

However, to determine the cost/benefit ratio of a particular optimization, we need to know how often the optimized region, and other regions whose running time may be affected by the optimization, will be executed during the program run. This requires knowledge of a program's dynamic behavior—information typically unavailable at compile-time.

Profile-driven compilation has been used as a partial solution to this problem. A program is compiled in a relatively unoptimized form, but instrumentation is inserted so that the program will record information about which regions of code are being executed. The program is then run on a set of training inputs, producing a *profile* summarizing the program's execution trace. The information in this profile is then used in a second compilation pass. Profile-driven compilation has been shown to be effective in many cases at managing the tradeoffs involved in code-optimization [5,27], making optimizations more aggressive than might otherwise be possible [15], and allowing optimizations based purely upon temporal information provided by the profile [13].

There are, however, significant drawbacks to profile-driven compilation. A common criticism is that the profiling cycle is awkward for developers. This is especially true for interactive applications where it is difficult to automate profile collection.

Although profile-driven compilation allows the compiler to make more intelligent tradeoff decisions, it fails to address the static nature of these decisions. Once made, they persist throughout the lifetime of the program's use. This places a great onus on the developer to profile his or her program on an input set representative of all possible ways in which the program may be used. In many cases, such an input set is difficult, if not completely impossible, to produce.

Furthermore, even if such a data set is found, profile-driven compilation still fails to capture the highly variant nature of program execution. Different runs of a program, or even different time slices within a single run, may exhibit widely varying dynamic behavior. At its best, profile-driven compilation represents optimization for the aggregate of all such cases. Ideally, however, a program would also be able to adapt in order to take advantage of the optimization opportunities offered by the dynamic nature of a particular run.

1.2 Dynamic Code Optimization

The central theme of this thesis is the idea of giving programs the ability to perform their own code optimizations at run-time. Rather than performing all code optimizations at compile-time, we investigate the effectiveness of *dynamic optimization*. In addition to performing static code optimizations, the compiler enables the continuation of the optimization process by generating an executable capable of monitoring itself and performing its own optimizations at run time.

There are two main run-time components of a dynamic optimization system: the profiler and the optimizer. The profiler collects information about dynamic execution behavior and uses heuristics to predict future behavior on the basis of past execution. The results of these predictions are given to the optimizer, which performs code optimizations to take advantage of anticipated patterns. This profiler-optimizer sequence is performed repeatedly over the lifetime of the program run.

Dynamic optimization offers a potential solution to the aforementioned problems of static optimization. At run time, all the program code, including all source files and dynamically linked libraries, is accessible in code space. Also, function boundaries no longer present as much of a challenge to optimization since machine code makes much less of a distinction between procedures than do high-level languages or compiler intermediate representations. As we will show, the development of interprocedural optimizations is not difficult in a dynamic optimization system.

Dynamic optimization also addresses the problem of inflexibility present in static optimization strategies. Run-time profiling allows the program to make optimization decisions based directly on current program behavior. These decisions can change as program behavior changes, both by undoing previous optimizations and by performing new ones. With dynamic optimization, the compiler does not even need to make decisions about which optimizations *might* be useful; if optimization routines are implemented in a dynamically linked library, they can be updated separately from the application code. Thus a long-since compiled program could expect its performance to improve as new hardware, profiling, and optimization techniques are developed.

There are, however, many potential pitfalls in the development of an effective dynamic optimization system. For instance, the run-time overhead of the system should not outweigh the optimization benefits. Also, it is not always easy to detect execution patterns early enough to exploit them. We need to make sure that the profiler does not take so long to detect interesting program behavior that we lose the ability to capitalize on this behavior while it is still relevant. This thesis represents a preliminary exploration of dynamic optimization and paves the way for more complete answers to these questions in the future.

1.3 Contributions

This thesis presents Deco, an automatic and selective dynamic optimization system we have built. This thesis makes the following contributions:

- It puts forth the idea of dynamic optimization: the run-time selection and optimization of program regions to take advantage of current program behavior. It points out the potential benefits and pitfalls of this technique.
- It identifies and explores the key issues involved in dynamic optimization: the identification of interesting program regions and the optimization of these regions.
- It introduces Deco, a working dynamic optimization system.
- It presents techniques and algorithms for dealing with some of the key dynamic optimization issues. We present algorithms for both region selection and optimization.
- It gathers preliminary results from Deco, and uses them to evaluate the effectiveness of these techniques and algorithms.
- By identifying the current strengths and weaknesses of Deco, it lays the groundwork for future work in dynamic optimization.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 presents a framework for profiling dynamic program behavior and uses this framework to give further motivation for dynamic optimization. Chapter 3 introduces the design and implementation of the current Deco prototype. Chapter 4 examines techniques for identifying at run time which areas of the program to optimize. Chapter 5 and Chapter 6 discuss efficient optimizations we have designed for use in the Deco run-time optimizer. Chapter 7 presents some results from the current Deco prototype. Chapter 8 discusses related work, and Chapter 9 summarizes our findings and discusses what future work should be done to extend the current Deco system. A glossary appears following the main body of this thesis to define important concepts whose definitions do not appear in the text.

Chapter 2 Analyzing Dynamic Program Behavior

This chapter focuses on how the run-time profiler collects information about dynamic program behavior. We present a profiling system for monitoring program execution, and argue why the model of dynamic behavior used by this profiler is useful for our purposes. We then use this model to analyze the behavior of a benchmark program in order to further motivate the need for dynamic optimization.

2.1 Path Profile Collection

A general rule of thumb used in program optimization is the *80-20 rule*, which states that a program spends 80% of its time executing 20% of its static instructions. The main job of the run-time profiler is therefore to identify this “hot” 20% of the code. Ideally, since the main object of finding the hot code is to optimize it, the profiler would identify the hot code in units conducive to optimization. Since code optimizations are written to work on groups of static instructions, what we really want is for the run-time profiler to pass the optimizer a *region*—a static instruction grouping which accurately reflects the program’s dynamic behavior.

How do we go about constructing a region? One well-documented type of region is a *path.* A path is a record of how a program’s control flow passes through a sequence of consecutively executed basic blocks [4]. Although by this definition the entire trace of a program’s execution defines but a single path, it is much more practical to subdivide this into shorter paths. *Path profiling* is the general term for profiling systems that summarize

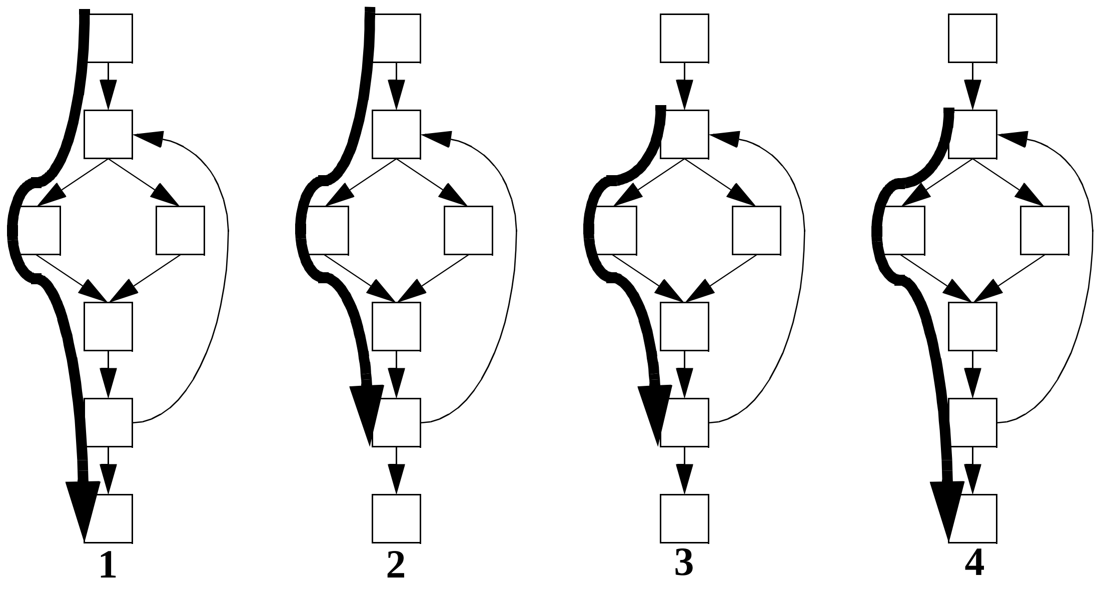

Figure 1. The four possible types of Ball-Larus paths, illustrated on a sample CFG. The top node is the ENTRY node and the bottom node is the EXIT node.

dynamic program behavior by producing either a temporal trace of path executions or aggregate data summarizing path execution behavior.

There are several different methods for path profiling, which differ primarily in their definition of how long a path should be. The path profiling method we have chosen for our system is drawn from the work of Ball and Larus [3]. The remainder of this section will provide a brief summary of the important points of this method, and attempt to justify it as a useful tool for describing program behavior. In an effort to disambiguate the meaning of the word “path,” we will refer to the kind of paths described here as “Ball-Larus paths.”

A Ball-Larus path is an intraprocedural, acyclic path. A procedure’s control-flow graph (CFG) can have four possible types of Ball-Larus paths (see Figure 1 for an illustration):

1. A path from the *ENTRY* node to the *EXIT* node.
2. A path from the *ENTRY* node to any node that is the source of a backedge.
3. A path from any node which is a backedge target to any node which is a backedge source.
4. A path from any node which is a backedge target to the *EXIT* node.

Because a given procedure's CFG has only a finite number of Ball-Larus paths, each Ball-Larus path can be uniquely identified by a single number. Weights can be assigned to the edges of the CFG so that the sum of the weights of the edges along each possible Ball-Larus path is unique. Numbering Ball-Larus paths in this way not only gives us a succinct identifier for each path, but also makes it very easy to reconstruct a path given its number [3].

Returning to our original concept of an optimization region, we can observe that Ball-Larus paths have many properties which make them particularly useful regions. First of all, the execution of a single Ball-Larus path concisely records the execution of many individual instructions and the direction of many individual branches. Since Ball-Larus paths never include a backedge, they are especially convenient for providing a record of execution within loop bodies, the places where programs spend most of their time.

Furthermore, Ball-Larus paths are simple building blocks from which we can construct more complicated regions. For example, if we find that a particular loop body executes two or more Ball-Larus paths with comparable frequency, it is fairly straightforward to construct static code representing the union of multiple paths—just lay out the subgraph of the CFG which includes all basic blocks that occur in any of the Ball-Larus paths. Similarly, if we wish to optimize a *superpath*—a sequence of Ball-Larus paths which frequently execute consecutively—we simply lay down in temporal order the Ball-Larus paths that constitute the superpath.

What makes Ball-Larus paths a realistic starting point for a run-time profiling system is that they provide all of this with relatively low instrumentation overhead. Information about the execution of all Ball-Larus paths can be collected using only a single machine register, which we will call the *path register*, and an associated data structure. The path register is zeroed before the start of any Ball-Larus path (i.e. on the execution of any backedge or on entry to any procedure). As the path is executed, the path register is incremented at certain points such that when the path terminates (i.e. when a backedge is taken or a procedure is exited) the path register has a value equal to the number corresponding to that path. The path register usually can be incremented in a single instruction, and an algorithm exists to minimize the number of these increments [3]. At the termination point of the path, code is inserted to record the execution of the path in the associated data structure.

Overall, Ball-Larus path profiling incurs an overhead of between 17% and 97% on various SPECint benchmarks [3]. The overhead incurred by the Deco implementation of Ball-Larus path profiling is somewhat higher than this, for reasons that will be explored in Chapter 4. Although not unrealistic for static profiling, this overhead is clearly much too high for a realistic run-time profiling system. Potential methods to drastically reduce profiling overhead will be discussed in Chapter 9.

2.2 Phased Execution Behavior

We stated in Chapter 1 that different time slices of a program run may exhibit different patterns of execution behavior. We refer to this phenomenon as *phased execution behavior*. This section uses path profiling as a tool to study the idea of phased execution behavior. We present an example of phased execution behavior in `povray`, a public-domain ray-tracing program, and briefly discuss how dynamic optimization might be beneficial in such a situation. Ray-tracing programs seem like prime candidates to benefit from dynamic optimization, since they are long-running, compute-bound applications.

Using the UNIX profiling tool `gprof`, we identified a heavily-executed function within `povray`. We then instrumented this function for Ball-Larus path profiling, and collected a path execution trace from a single program run. We further narrowed our focus by looking at only the first million path executions (which we will call “phase one”) and the last million path executions (“phase two”).

We identified a total of six superpaths that were frequently executed in phase one or phase two. These superpaths varied in length from three to six Ball-Larus paths. For each superpath, if we multiply its length by its frequency within a phase, we get a count of how many path executions in the phase occur as part of the given superpath.

Using this metric, we find that the execution of these superpaths constitutes a high proportion of the total number of path executions in both phases. In phase one, these superpaths comprise 96% of the total number of path executions; in phase two, they account for

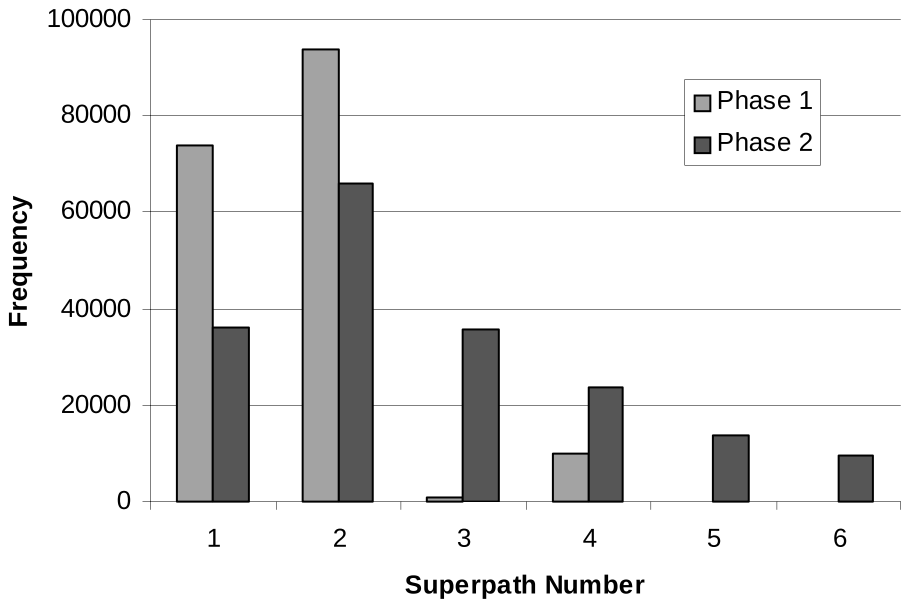

Figure 2. Superpath execution frequencies in phase one and phase two.

87% of total path executions. Although we are not adjusting for path length, we can nevertheless be confident that we have picked a set of superpaths which is very significant to the execution time of both phases.

Figure 2 shows how the frequency of each superpath varies in each phase. While phase one executes superpaths 1 and 2 almost exclusively, phase two shows a more varied distribution of superpath execution. For example, superpath 3 barely occurs at all in phase one but has a very high execution frequency in phase two. These differences in superpath execution frequency imply that the same optimization strategy might not be desirable for both phases.

Assume that since the superpaths overlap, it may not be possible to optimize them all simultaneously. For instance, if we can only optimize either superpath 2 or superpath 3, which one should we choose? Looking at both phases collectively, we would probably choose to optimize superpath 2, since it occurs with high frequency in both phases. But suppose that superpath 3 is much more amenable to optimization than superpath 2. Superpath 2 would still be a better choice for phase one, but phase two might run faster if we optimized superpath 3 instead.

Dynamic optimization gives us the potential to detect and take advantage of phased execution behavior within a program run. A static optimizer would have to choose either superpath 2 or superpath 3, since it has no way of changing its decision once the program is compiled. However, a dynamic optimizer could evolve as program behavior changes, optimizing superpath 2 during phase one and superpath 3 during phase two.

Chapter 3 The Design of Deco

This chapter gives a high-level overview of Deco, our system for dynamic optimization. This overview will trace the lifetime of a Deco-ized program, from source code through a complete program run, giving a brief description of each part of the Deco system. Later chapters will provide more specific details about the novel components of this system.

Deco has been designed primarily as a platform for the exploration of the dynamic optimization problem space. Therefore, the design emphasis has been on modularity and flexibility. Although the current prototype optimizes Digital Alpha executables, we have designed the system so that it is easy to port to a new machine or even an entirely different target architecture. We have not yet devoted a great deal of effort towards minimizing the costs of the individual run-time components. Suggestions for such improvements appear in Chapter 9.

3.1 Characteristics of an Optimization Region

The goal of the current system is to dynamically identify and optimize simple interprocedural regions that are executed frequently. Because programs spend almost all of their time executing loops, we limit our initial selection of hot regions to Ball-Larus paths that represent loop bodies.

However, our region formation does not stop there; simply optimizing intraprocedural loop bodies would severely inhibit our optimization opportunities. Hank et al. present statistics which show that a significant portion of an application's execution can look as though it was spent in acyclic code if one limits one's view by procedure boundaries [16]. This data implies that programs spend a great deal of time in interprocedural loops.

To take advantage of this, we perform *partial inlining* on the paths we select for optimization. If a hot Ball-Larus path is found that contains a function call, the system attempts to identify a hot entry-to-exit Ball-Larus path in the called procedure. If such a path is found, it is inlined into the code for the outer Ball-Larus path in the caller, so that the entire interprocedural code sequence can be optimized as a single static instruction list. This type of inlining is called "partial" to distinguish it from the inlining of an entire procedure body. Deco can perform recursive partial inlining, enabling it to expose and optimize loop bodies that span several procedures.

3.2 Compilation of Deco Executables

Deco is functionally divided into two parts: the compilation component and the run-time component. The compilation component is responsible for inserting instrumentation code to enable run-time collection of path profiles and for generating dataflow analysis information that will later be used by the run-time optimizer. It is implemented as a set of passes written within the framework of the SUIF 1.1.2 compiler infrastructure [25]. These passes are run as part of a complete SUIF compilation sequence from C into assembly language (see Figure 3).

The *label* pass implements the algorithms for Ball-Larus path profiling. Considering the CFG of each procedure separately, it computes minimum-impact locations within that CFG in which to place instrumentation for updates to both the path register and the profiling data structures. It marks these locations with SUIF annotations on the appropriate CFG edges.

Also, for each procedure CFG, the label pass outputs a `which_way` function, a C function that provides an interface to decode a Ball-Larus path at run time. If we pass a unique Ball-Larus path number to the `which_way` function, it provides us with the static list of instructions representing that Ball-Larus path. The label pass also creates a dynamically-accessible data structure that maps procedures to their corresponding `which_way` functions. The `which_way` functions thus provide a simple means for the run-time optimizer

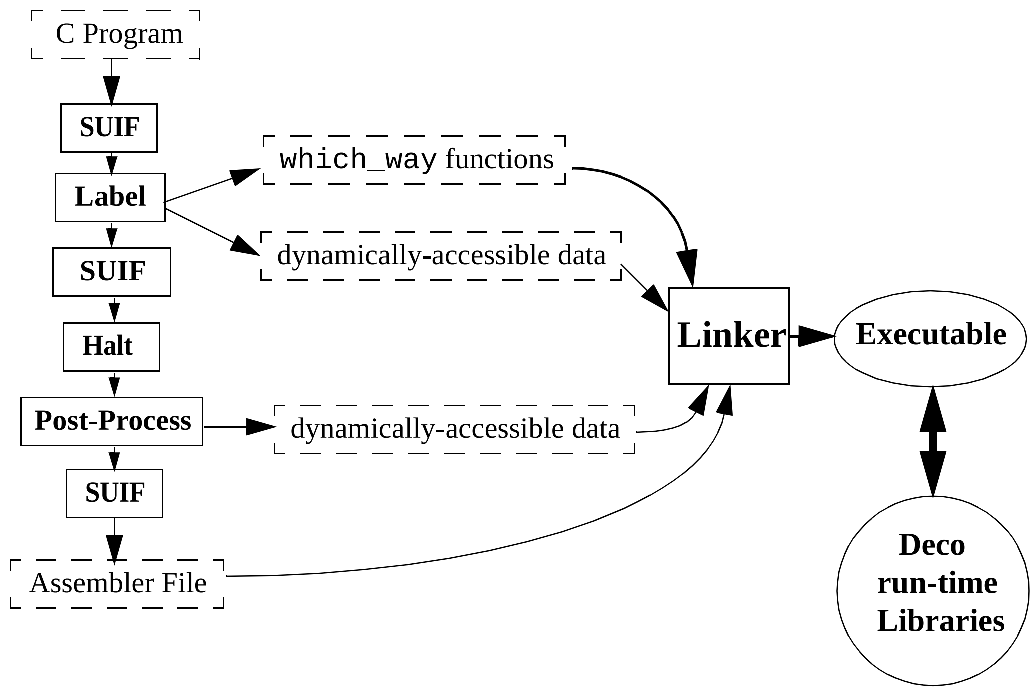

Figure 3. The compilation of a Deco executable. The boxes labeled **SUIF** *represent sequences of one or more standard SUIF passes.*

to decode frequently-executed Ball-Larus paths. The C code for the `which_way` functions is compiled into the final executable.

The next Deco pass is a slight modification of the SUIF *halt* pass [28]. This pass looks for the annotations inserted by the label pass and inserts instrumentation code at the appropriate points. There are two kinds of instrumentation points: *lightweight instrumentation points*, which only require an update to the path register, and *heavyweight instrumentation points*, which require resetting the path register and updating the associated profiling data structures. At lightweight instrumentation points, the halt pass inlines the one instruction necessary to increment the path register by the appropriate amount. (Occasionally, more than one instruction is necessary, if the increment sequence requires loading a large immediate value.) The path register is simply a register we have reserved for this purpose; it is callee-saved so that it will not be corrupted by calls. At heavyweight instrumentation points, the halt pass inserts a function call to update the profiling data structure; the code for this function exists in a shared library that will be linked in at run time. It also inlines an instruction to reset the path register.

The final compilation pass is the *post-processing* pass. This pass does not actually modify the executable; its job is simply to collect dataflow information for use by the run-time optimizer. It performs two main functions. First of all, it computes register liveness information for a subset of the program's basic blocks. Secondly, for each function call instruction in the executable code, it records which argument registers are used by the call and which return registers are defined by the call. All of this information is recorded in dynamically accessible data structures to make it easily available to the run-time optimizer. We call this process of making compile-time information available at run-time *hint passing*.

Note that since instrumentation and hint passing are performed by the compiler, not all program procedures are affected. Those procedures in dynamically-linked libraries, for instance, will lack profile information. This is certainly a drawback to the current implementation. One solution to this would be to compile all dynamically-linked libraries, such as the standard C libraries, with Deco (although we do not do this in the current system). Another possible solution to this problem is presented in Chapter 9.

3.3 The Run-time Profiler

All of the work done during compilation sets the stage for the run-time optimization system, which is the core of Deco. Aside from the `which_way` functions, which are compiled in as C code (hint passing, which is described later, also produces some C code which is compiled into the executable), the run-time Deco routines are in shared libraries, linked with the executable at run-time.

A result of this arrangement is that the run-time optimizer, and the optimized code it produces, share a single address space with the original executable. The benefit of this approach is that a transferring between the original code and the optimizer or optimized code requires only a simple branch or jump—there is no overhead of a context switch. This provides us with the potential to achieve performance wins even on relatively small code regions.

The interface between the original executable and the Deco run-time library is accomplished through the procedure calls inserted by the halt pass at the heavyweight instrumentation points. Every time a Ball-Larus path completes execution, a call is made into the *run-time profiler*, which records the path execution in an internal data structure.

The *trip mechanism* runs on top of the run-time profiler. It keeps track of the activity of the profiler, and tries to predict future behavior based on the path execution information that the profiler records. Exactly what heuristics the trip mechanism should employ in performing this task is an open research question. Our current approach is detailed in Chapter 4, and ideas for other trip mechanisms are described in Chapter 9.

When the trip mechanism detects a potentially useful pattern in the program's execution, it "trips", invoking the run-time optimizer and passing to the optimizer an identifier for the region it thinks should be optimized. Since the current implementation of the trip mechanism trips on single Ball-Larus paths, the region identifier currently consists of a procedure number plus a Ball-Larus path number. Because of the phased nature of program execution, we would expect the trip mechanism to trip many times during the course of a single program run, as it continually tries to adapt its estimate of the best optimization strategies.

3.4 The Run-time Optimizer

The *run-time optimizer* takes a region identifier as input and optimizes the execution of this region. It does this by outputting an optimized version of the region somewhere else in code space (called the *code optimization space*, or *cos*) and overwriting the first instruction of the hot path in the original code with a branch into the cos. Because it minimizes modifications to the original code, this model allows for a great deal of flexibility in creating and removing optimized code.

The various components of the run-time optimizer are illustrated in Figure 4. The first component is the *decoder*, which uses the region identifier to produce a set of static instructions corresponding to the region we want to optimize. With the help of the `which_way` functions, the decoder walks through the code space of the executable, retracing the sequence of machine instructions that constitute the given Ball-Larus path. It stops when it reaches the backedge of the loop, and therefore does not copy the code on the backedge for the heavyweight instrumentation point. Since we want our optimized code to

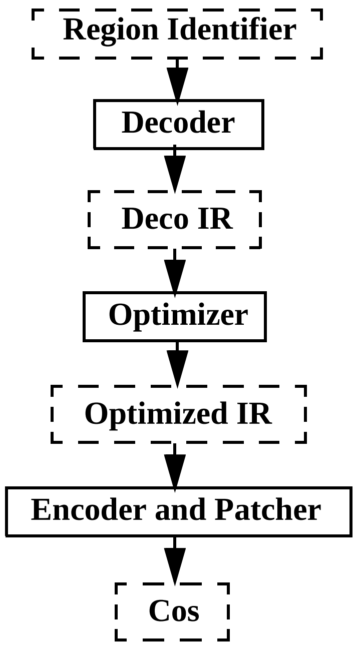

Figure 4. The phases of the Deco optimizer (solid boxes) and their outputs (dashed boxes).

run as quickly as possible, it is important to remove the calls to the instrumentation routines.

The decoder also performs partial inlining. As was previously mentioned, when the decoder encounters a call instruction, it queries the run-time profiler to determine if there is a hot entry-exit Ball-Larus path in the called procedure. “Hotness” in this context is a threshold on the percentage of times a particular entry-exit path is taken relative to the set of all entry-exit paths. If there is a path that exceeds the threshold, the decoder ignores the call, and instead walks the hot entry-exit path, returning to the original Ball-Larus path only when it reaches a `return` statement. This process continues recursively.

Binary instructions are difficult for the optimizer to deal with; interpreting the bit fields is a cumbersome process. Therefore, to facilitate later optimization, the decoder converts each binary instruction it encounters into Deco’s *intermediate representation* (IR). The Deco IR is a set of C++ classes which represent operands and instructions. Like a compiler IR, the Deco IR provides interface methods for dealing with code objects, abstracting away the machine-specific details. The IR enables the optimizer to get quick answers to queries about instructions (“Can this instruction change the value of the program counter?", "What are the source operands of this instruction?") and operands ("Is this operand a register?" "What is the value of this immediate operand?"). The Deco IR has been written with the same interface as the IR for Machine SUIF 2.0, so that optimizations written for one system can be ported to the other with minimal changes.

The Deco IR also captures the control flow of the selected path. Because the trip mechanism trips on Ball-Larus paths within loop bodies, the kind of hot region we currently identify is a sequence of consecutively-executed basic blocks, ending in a backedge to the top of the sequence. Therefore, the IR can capture the basic control flow of the hot region by representing it as a single block of instructions with an edge from the bottom of the block to the top.

However, this simple representation is not quite sufficient, since it assumes we will remain on the hot path forever (i.e. run *ad infinitum* along the same path through the same loop.) This is clearly insufficient. Executing the basic block at the top of the hot path does not imply that we will then proceed to execute every block in the hot path. Any non-trivial Ball-Larus path contains conditional branches, and, because of the definition of a Ball-Larus path, only one of the two possible branch targets is included on the path. Although we would hope to stay on the hot path for a long period of time, we will most likely eventually execute an iteration which "falls off" before reaching the end, because the dynamic direction of some branch didn't match our predictions.

In order to maintain consistency with the original code, the IR must cope with the fact that we can fall off the hot path. Our hot path is really a *superblock*—a single-entry, multiple-exit, linear code sequence [19]. In the event that we fall off the hot path, we want to return control flow to the original code.

However, we may have to "fix up" some program state before returning. For example, if our hot path contains partially inlined code sequences, executing in the hot path will cross procedure boundaries without properly setting the return address register (since we do not execute a `call` instruction before entering the inlined code.) Therefore, if we fall off the hot path within one of the inlined code sequences, we need to reset the return address register to have the correct value. (The correct value in this case would be the

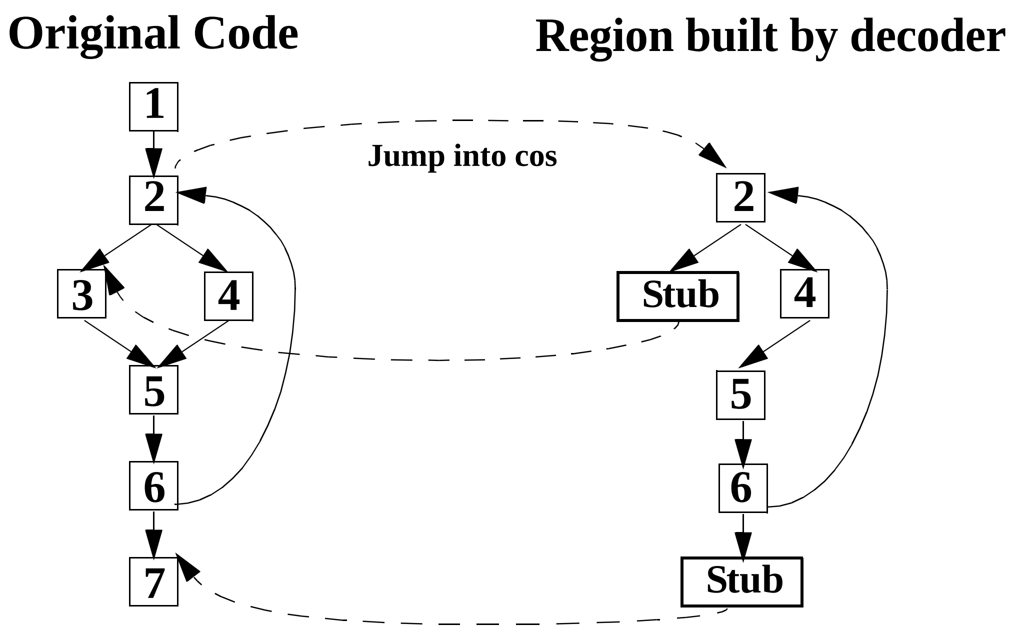

Figure 5. An illustration of a simple path pulled out by the decoder, showing the location of stubs and the control flow relative to the original code.

address of the instruction after the `call` instruction in the original code that would have gotten us into the procedure from which we extracted the partially inlined path.) This is only one of a number of similar bookkeeping issues. Running optimizations, which changes the instruction sequence, dramatically increases the amount of fixing-up we have to do before returning to the original code.

In order to deal with this problem cleanly, the IR contains *stubs*, basic blocks containing compensation code to fix up program state in the event we fall off the path. The IR associates a stub with each instruction that could change the program counter unexpectedly, causing us to fall off the path (i.e. conditional branches and indirect jumps). We initialize each stub with all of the necessary compensation code to fix up program state if we fall off the path at the given point (later optimizations may add additional stub code). The last instruction in the stub is a branch into the original code.

We realign all conditional branches so that the fallthrough target of each branch stays on the path and the taken target brings us to the top of the associated stub. (The way we handle indirect jumps is a bit more complex, and will not be discussed here.) Thus, any time we fall off the path, we will always execute the compensation code before returning to the original code. The idea for stubs is drawn from Cohn and Lowney’s work on hot-cold optimization [7]. Figure 5 shows an example hot region built by the decoder.

Once we have built the hot path in the IR, we are ready to perform our various code optimizations. These optimizations transform the IR, often shortening the hot sequence. The specific set of optimzations performed in the current Deco prototype is discussed in Chapter 5 and Chapter 6. It is the responsibility of the optimizations to make sure that the stubs continue to contain correct compensation code to fix up whatever the optimization does to the hot path. The output of the optimizer is a transformed IR.

Once we have performed all the optimizations we want to do, we pass the IR to the *encoder*. As its name implies, the encoder is the opposite of the decoder, turning the Deco IR into a corresponding set of machine instructions. These machine instructions are then written into a cos.

Since the cos is located in code space (in a section we reserve specifically for this purpose), it should be close enough to the original code to be reachable by a single unconditional branch instruction (as opposed to an indirect jump). The *patcher* overwrites the first instruction on the optimized path in the original code with a branch to the cos.

The run-time optimizer has now completed its work. Control flow returns to the heavyweight instrumentation point which caused the trip mechanism to invoke the optimizer. Until we unpatch the original code, whenever we reach the first basic block in the now-optimized path, we will branch into the cos and continue execution in the optimized version of the code until we fall off.

Chapter 4 Identification of Hot Paths

Clearly, the success of a dynamic optimization system such as Deco depends most heavily on selecting good regions to optimize. We want to find regions which will be executed frequently and will be amenable to optimization. This chapter discusses how the current Deco prototype deals with these issues. It provides a more in-depth look at the algorithms used by the run-time profiler and the trip mechanism.

4.1 Recording Path Executions

Even in a moderately sized application, the number of possible Ball-Larus paths can be extremely large. Statically allocating a data structure to record information about the execution of each and every possible path is therefore impossible. Fortunately, however, we have observed that the subset of possible paths which are actually executed over the course of any given program run is actually quite small. We have thus designed our path profiling data structure to scale dynamically to the number of paths executed at run-time.

The halt pass inserts a call to a Deco *initialization function* at the very beginning of the program's `main` procedure. The initialization function itself resides in one of the Deco shared libraries. The initialization function allocates an array equal in size to the number of instrumented procedures in the program. Each element in the array is a hash table, and an entry in the hash table contains information about a particular Ball-Larus path in the procedure (its unique number and execution count, for instance). The hash table is indexed by Ball-Larus path numbers; collisions are resolved by chaining. The hash table is initially empty, and we fill it in dynamically. We do not put a path into the hash table and begin recording information about it until we have actually seen it execute at least once during the program run. This entire data structure is referred to as the *profile table*.

The drawback of the profile table is that it increases the profiling overhead (relative to a simpler data structure such as an array.) There is a small cost for computing the hash function on the Ball-Larus path number, and there is also a penalty associated with seeing any path execute for the first time. Furthermore, for modularity, the current implementation of the profile table includes a fair amount of procedure call overhead. A much more efficient implementation could be achieved by inlining the appropriate code at the heavy-weight instrumentation points.

Recall that the path execution information recorded by the run-time profiler is used for two separate tasks. First, it is used by the trip mechanism to select a hot loop-body path. Secondly, it is used by the partial inliner to select hot entry-exit paths. It thus makes sense for the run-time profiler to treat backedge-terminated and exit-terminated paths differently. Entries in the profile table contain a boolean field to note whether a path is backedge- or exit-terminated.

When trying to select hot loop-body paths, it is important to find a path which is not only executed frequently, but is also likely to be executed many times in a row. We would like to iterate through the backedge of the optimized loop path as many times as possible before falling off path. This allows us to amortize the cost of the compensation code in the stubs, which we hope will be far outweighed by the cumulative benefit of executing the optimized loop many times.

We thus attempt to select paths that will have this property when we optimize them. The run-time profiler records the execution of a backedge-terminated Ball-Larus path only if this execution directly follows an execution of the same path. Therefore, the execution counts in the profile table for backedge-terminated paths are representative of how often the path followed itself in consecutive iterations of the loop.

However, for the inlining of entry-exit paths, we only care that we are selecting a highly probable path—there is no issue of consecutive iteration. Therefore, the execution counts for exit-terminated paths are true execution counts, representing the number of times we have seen the particular path execute.

It is vital that we record statistics for all exit-terminated paths, not just entry-exit paths. Recall that not all exit-terminated paths begin at procedure entry; they can also begin at backedge targets. When we partially inline an entry-exit path, we want to make sure that not only was it the most likely entry-exit path, but it was also the most likely of all paths to the exit node. Including information about all exit-terminated paths takes into account that some of the procedures we might wish to partially inline have loops in them, and are thus unlikely to have simple acyclic paths amenable to partial inlining.

4.2 The Continuous Monitoring Trip Mechanism

The trip mechanism uses the information recorded in the profile table to select a good path for optimization. There are two basic execution models one might use for a trip mechanism. The first is the *continuous monitoring* model, where the trip mechanism observes all the activity of the run-time profiler, and trips when it thinks it sees something interesting.

A previous version of Deco used a very simple continuous monitoring scheme. At compile time a *trip count* was initialized. Every time the run-time profiler updated an entry in the profile table, the trip mechanism checked to see if the execution count of the given path exceeded the trip count. If it did, the path identifier was passed to the run-time optimizer. We implemented this as a quick and simple mechanism to test the early prototype—we never expected it to be particularly effective (and we were not pleasantly surprised). However, the reasons for its ineffectiveness are enlightening in that they show us desirable properties of a good trip mechanism.

There were two main problems with this trip mechanism. First of all, the value of the trip count was arbitrary. What we would really have liked is for the trip count to have been in some way related to the total length of the program run. That way, if something tripped, we would know it constituted a decently high percentage of program execution time. Unfortunately, knowing the length of a program run ahead of time would imply a solution to the halting problem (which has been proven to have no computable solution), and is therefore impossible.

A continuous monitoring system could better address this problem if, in addition to recording a particular path’s execution count, it kept track of the ratio of the execution count for that path to the total execution count of all backedge-terminated paths. We could then trip when this percentage reached a certain level. Of course, we would want to allow such a system some “warm-up” time—otherwise we would always trip on the first path we execute. A drawback of this system is that we have to update these percentages every time we execute a path, which adds quite a bit to our instrumentation overhead.

The second problem with the trip count method is that it does not accommodate program phases. Under the trip count model, something is either hot or it isn’t; there’s no concept of something being hot only during certain periods of time. This is also a problem with the slightly improved continuous monitoring system described in the above paragraph.

4.3 The Interrupt Trip Mechanism

This leads us into the second execution model for the trip mechanism, the *interrupt* model. Rather than continuously monitoring the activity of the run-time profiler, the interrupt model instead examines the profile table only at intervals, zeroing the execution counts after each examination. In this way, we effectively break the execution trace into smaller pieces, not allowing information about a path’s execution a long time ago to affect our view of its current hotness. An additional advantage of this model over the continuous monitoring model is that we reduce the total amount of computation.

The trick here is obviously to pick a good interval size. This problem is made easier by the fact that a program phase must be of a non-trivial length in order for it to be useful from an optimization standpoint. Therefore, if we set our interval size fairly small, we should be able to detect phases of at least moderate length fairly early in their execution. Our preliminary experiments seem to indicate that the exact interval length is not nearly as important to an interrupt trip mechanism as the exact value of the trip count is to the trip count mechanism.

The current Deco system implements an interrupt model trip mechanism which attempts to approximate the behavior of the more ideal continuous monitoring system described above. Once a procedure has executed a certain total number of backedge-terminated paths (this number is set to be relatively low), the trip mechanism examines its entry in the profile table, calculating for each backedge-terminated path the ratio of its execution count to the total execution count of all backedge-terminated paths in the procedure. It then finds the most-likely path (i.e. the path with the highest ratio.) If the most-likely path's ratio exceeds a certain threshold, it is passed to the optimizer. The execution counts for the backedge-terminated paths of the examined procedure are then zeroed, so that another interval can begin.

Note that we are selecting our hot paths on a per-procedure, rather than a global, basis. At first glance, this might seem like a poor strategy, since we want to select globally hot paths. However, keep in mind that the interval length is the same for all procedures. Therefore, procedures which execute more often will attract the attention of the trip mechanism more frequently. This means that most of the paths we pull out will be globally frequent in addition to being locally hot.

As we will show in Chapter 7, this trip mechanism seems to work fairly well for many of our benchmarks. Clearly, however, the design of a good model for a trip mechanism, as well as the proper parametrization of that model, is an area in need of further research.

4.4 Multiple Coses

The trip mechanism will identify many paths as hot over the course of the program run. The optimizer will then optimize each hot path and put it into a cos. However, the idea of having multiple optimized coses in the program at the same time introduces a number of issues that need to be addressed.

First of all, we may not want too many coses around at the same time. We would like to keep the growth of our optimized code in check, since code size expansion can be detrimental to performance. Secondly, cos entry points can overlap: multiple Ball-Larus paths in the same procedure will share the same first basic block. The patcher can only overwrite the first instruction in the block with a branch to one or the other: which should it choose? Thirdly, even if we could keep every cos around at the same time, the idea of phased execution behavior argues that we should not do so. If a hot path trips close to the beginning of the run, is it still going to be a hot path at the end? If we want to optimize some other path with the same entry point, the presence of this stale path will inhibit us from doing that.

We attempt to solve these problems by introducing the idea of a *cos cache*. At the beginning of the program run, Deco reserves a fixed amount of memory right after the text section of the program. It divides this memory up into a relatively small number of equal-sized blocks. (The block size is set to a number of bytes larger than the longest cos we have ever observed in testing the Deco system.) Each of these blocks will hold exactly one cos at any given time. Thus, only a fixed number of coses exist concurrently. The cos cache is fully associative: any block can hold any cos. When a new cos is introduced, we first attempt to place it in an empty block in the cos cache. If no such empty block exists, we randomly evict one of the current coses from the cache in order to make room for the new cos.

The only catch here is that we cannot evict a cos that is on the current call stack. Since the current implementation trips on a per-procedure basis, it is possible that the following sequence of events occurs:

1. We create cos A.
2. We begin executing in cos A, and reach a call statement (since not all calls are inlined).
3. We execute the call, and before returning from it, some other path trips, creating cos B.

Were we to now evict cos A to make room for cos B, the call from cos A would return to find the code following it completely changed. This will cause incorrect program behavior.

We solve this problem by associating a reference count with each cos. We add a preheader to the loop inside the cos, so that when we enter the cos, we increment the reference count; when we leave the cos along one of the stubs, we decrement the reference count. We do not evict any cos which has a non-zero reference count. The only extra code introduced here is in the preheader and the stub, which execute only once each; the impact on the running time of the cos is therefore minimal.

The problem with evicting coses on the current call stack imposes an effective minimum on the size of the cos cache. We want to make sure our cos cache is large enough to make it highly probable that at least one cos will not currently be executing, allowing us to evict it. Although making the cos cache too small would not cause incorrect program behavior, it would be highly detrimental, because it would not allow us to optimize all the paths recommended by the trip mechanism.

The cos cache helps to solve some of the problems of multiple coses. We ensure that the total memory requirement of all the coses is constant throughout the run of the program. Also, eviction provides a very natural way to “retire” a cos when its phase is over. A better eviction policy might be to evict the least-frequently-used cos. However, the random eviction policy is not necessarily that bad. If we make a mistake, the trip mechanism will correct it by tripping on the evicted hot path once again.

The cos cache must also be modified to address the problem of multiple hot paths with the same patch point. To simplify the identification of situations where this is an issue, the current Deco implementation makes the overly-conservative assumption that any two paths in the same procedure will have the same patch point. If the trip mechanism finds a new hot path in a procedure for which there is already a cos, we want to give preference to the new path, since it represents a new phase in local behavior. We therefore evict the old cos for this procedure, replacing it with the new one.

There is one problem with the above solution. Since a cos doesn’t contain any heavyweight instrumentation on its backedge, the run-time profiler doesn’t record executions of the cos. Therefore, what the trip mechanism finds by looking at the profile table is actually the hottest path only if we ignore whatever hot path we might already have pulled out from this procedure. Optimizing this new hot path would evict the current cos for this procedure, which is only a good thing to do if the new path is taken more frequently than the cos.

To solve this problem, we add a bit of additional code to the cos to record the execution count of its loop. The impact on the execution time of the cos loop is minimal; most of the new code is placed in the preheader and stubs and only a single instruction is added to the loop path. We zero out the cos execution count every time the trip mechanism examines the procedure from which the cos path is taken.

We only replace the cos for a procedure with a new hot path if the new hot path had a greater execution count during the just-completed interval than hot path currently in the cos. In this way, we assure that the lack of profiling instrumentation in the cos does not have a detrimental effect on the selection of regions to optimize.

Chapter 5 Optimizations

Once we have selected good regions, the onus of actually speeding up program execution falls on the optimization algorithms. When optimizing code at run time, the optimizer's running time is much more of a factor than it would be at compile time. We are interrupting program execution in order to optimize this section of code. Therefore, when evaluating a run-time optimization algorithm, the main metric one should use is the benefit/cost ratio. A corollary to this principle is that not all run-time optimizations are worthwhile. One should not waste valuable execution time doing work that could have been done just as well, or almost as well, at compile time.

This chapter outlines Deco's current optimization suite. We have developed a few algorithms specialized for optimizing the type of partial CFG the decoder produces. These optimizations are based somewhat on hot-cold optimization by Cohn and Lowney [7], although we have redefined and modified their work extensively to suit the particular needs and parameters of Deco. The current prototype does not perform very aggressive optimizations; the focus during the early development of the optimizer has been on fast, lightweight code transformations.

5.1 Hint Passing

We have implemented a generic, extensible hint passing system for Deco to allow the easy transfer of information from the compiler to the run-time optimizer. This system, which was described briefly in Chapter 3, epitomizes Deco's view of code optimization as a continuous process, rather than an operation limited to a specific time frame. This section will give a more complete overview of how we have implemented hint passing.

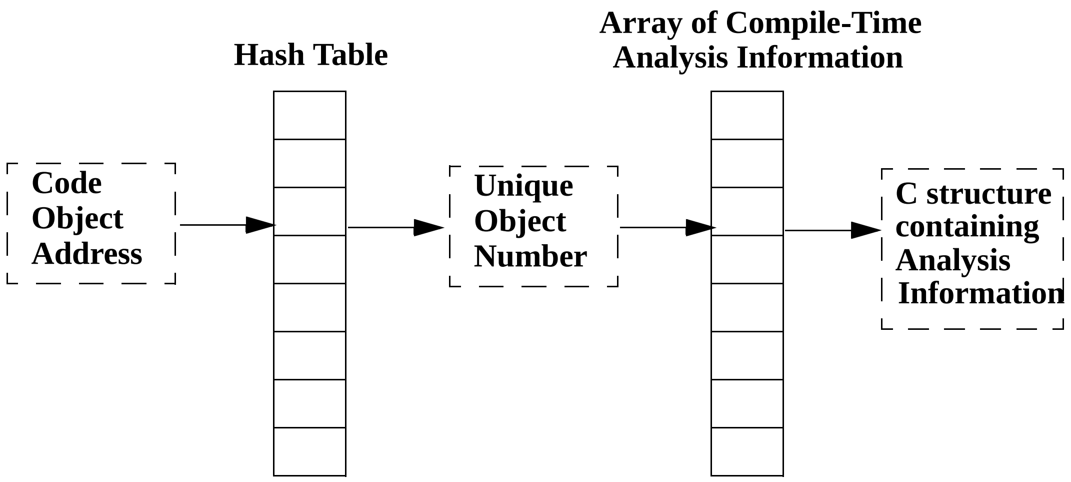
Hash Table
Array of Compile-Time Analysis Information
Code Object Address
Unique Object Number
C structure containing Analysis Information

Figure 6. The run-time data structure created by the hint-passing system. A separate instance of this structure exists for each type of code object.

Some of our optimizations require expensive dataflow analysis information, such as register liveness. This information is often difficult to compute at run-time, and we do not want to take up program execution time doing so. Therefore, it is much better to gather the information at compile-time and leave it in a form where it will be available to the run-time optimizer. Thus, from the standpoint of the run-time optimizer, it will appear as if dataflow analysis information is gathered instantaneously.

Figure 6 shows a diagram of our hint-passing interface; this interface is independent of the information being passed. The interface allows the compiler to uniquely number code objects (such as basic blocks) and associate with these object identifiers a C structure of analysis information. At compile time, the post-process pass generates a map (implemented as a hash table) from code object address to unique identifier. The post-process pass also generates a C routine that initializes both the hash tables and the associated tables of analysis information. This C routine is compiled in with the executable code, and is called by the Deco initialization function at the beginning of each program run. (The reason we create the hint-passing data structures at run time, as opposed to having the compiler make them static objects, is that the code object addresses not known at compile time.)

The run-time optimizer accesses the analysis information by means of a simple functional interface. Making calls to routines in the Deco run-time libraries, the run-time system can, in constant time, obtain the C structure with a code object's analysis information simply by specifying either the address or the unique identifier of the object.

What makes this system particularly useful in the Deco prototype is its extensibility. If we wish to modify the hint-passing system to collect a new piece of analysis information about an existing type of code object, we simply modify the data structure associated with that object. Similarly, if we wish to add a new type of code object to the system, we merely create a new analysis-information data structure for this code object.

The current hint-passing system keeps information about three types of code objects: procedures, call instructions, and basic blocks. It keeps the following information about each code object:

- Procedures: the address of the corresponding `which_way` function; some information to help us determine which Ball-Larus path numbers correspond to paths that begin at the entry node (which helps us find entry-exit paths for partial inlining.)
- Call instructions: the registers used and defined by the called function.
- Basic blocks: for each basic block that has a predecessor node ending in a conditional branch, we record which registers are live on entry to the block. Note that this class of basic blocks exactly describes all blocks which could be targets of cos stubs. Therefore, this liveness information can be translated into information about which registers are live when we fall off the cos.

5.2 Reaching-Definitions Analysis

Unfortunately, not all analysis information can be collected at compile time. The above pieces of information can, because they are all properties of static objects in the application code. However, some dataflow analyses compute dynamic pieces of information, and therefore need to be done at run-time, once we have determined the exact optimization region.

We have written our optimizations so that the only such run-time analysis we need to perform is *reaching-definitions* analysis [1,19,20]. A *definition* of a register `x` is an instruction which assigns a value to `x`. Suppose we have some definition `d` for `x`. For any instruction `i`, `d` is a *reaching definition* of `x` for `i` if and only if there exists a possible sequence of control flow from `d` to `i` along which there is no other definition of `x`. Intuitively, this means that `d` is one of the instructions responsible for setting the value that `x` contains when `i` is executed. Note that by this definition, `i` can have more than one reaching definition for `x`, since the program's dynamic control flow may be able to get to `i` in one of several different ways, each with its own reaching definition. Reaching-definitions analysis finds the reaching definitions for each register at each instruction.

Reaching-definitions analysis within a cos is simpler than it is within a procedure CFG. Since the cos only has a single control flow sequence, each instruction can have at most one reaching definition within the cos for each register. However, since the cos is only a piece of a complete CFG, it is possible that for some instruction-register pairs, reaching definitions will come from outside the cos.

For which operands do we want to compute reaching definitions information? The Alpha architecture has 64 registers, 32 integer registers and 32 floating-point registers. However, one register in each unit is reserved for holding the constant value of zero, and thus cannot be defined. This leaves us with a set of 62 interesting register operands.

We would also like to treat memory as an operand for reaching-definition purposes. We consider store instructions as "defining" memory. The current system makes no attempt to distinguish between different dynamic memory addresses. This gives us a total of 63 operands in which we are interested. However, we make our sets 64 elements long, because this size is more convenient in the 64-bit Alpha architecture.

For every instruction in our cos, we want to compute two sets of length 64:

- A *cos-reaching-definition set*, which, for each of our 64 operands, contains a pointer to the instruction in the cos that contains the reaching definition this operand. This pointer is set to `NULL` if there is no definition of this operand in the cos.
- An *outside-reaching-definition set*, which, for each of our 64 operands, contains a boolean indicating whether it is possible for a definition of this operand outside the cos to reach this instruction.

Again, note that it is possible for a given instruction to have both a cos reaching definition and an outside reaching definition for a particular operand. This is because it is possi-

```
def(i): the set of operands defined by i

/* Initialize the cur_cos_reaching_defs set */
for(j=0 to 63)
   cur_cos_reaching_defs[i] = NULL;

/* First iteration:
   Find the definitions that reach the bottom of the loop */
for(i=first_instruction to last_instruction)
{
   for(r in def(i))
      cur_cos_reaching_defs[r] = i;
}

/* Initialize the cur_outside_reaching_defs set */
for(j=0 to 63)
   cur_outside_reaching_defs[i] = TRUE;

/* Second iteration:
   Set the information for each instruction */
for(i=first_instruction to last_instruction)
{
   cos_reaching_definition_set[i] = cur_cos_reaching_defs;
   outside_reaching_definition_set[i] = cur_cos_outside_defs;
   for(r in def(i))
   {
      cur_cos_reaching_defs[r] = i;
      cur_outside_reaching_defs[r] = FALSE;
   }
}
```

Figure 7. The algorithm to compute the reaching-definition and outside-definition sets for each instruction in a cos.

ble to enter the top of the cos loop in two ways: either from the patch point, if we are just entering the cos, or along the cos backedge, if we have already been executing inside the cos. Consider, for example, an instruction `i` at the very beginning of the cos. If we are just entering the cos and executing `i` for the first time, the current value of `x` will have been set by some instruction outside the cos. However, if some later instruction in the loop defines register `x`, this definition can reach subsequent executions of `i` along the backedge.

We can compute the value of these two sets for every instruction with only two iterations over the cos loop. The pseudocode for this algorithm is shown in Figure 7. The first iteration computes which definitions in the cos reach the cos backedge. We begin by initializing a set of 64 instruction pointers called `cur_cos_reaching_defs` to contain all `NULL` pointers. We then iterate over the instructions in the cos loop. For each instruction, we determine its *definition set*, the set of all operands it defines. For each operand `r` in the definition set, we set `cur_cos_reaching_defs[r]` to point to the current instruction. When we finish iterating through the loop, `cur_cos_reaching_defs` will contain, for each operand `r,` the last instruction in the cos loop which defines `r`. This is exactly the set of definitions that reach the backedge of the loop.

Our second iteration over the loop will compute the exact cos-reaching-definition and outside-reaching-definition set for each instruction. Before performing this second iteration, we initialize a set of boolean values called `cur_outside_reaching_defs` to contain all `TRUE` values. We once again iterate over the instructions in the cos loop. For each instruction, we set its cos-reaching-definition set to be the current value of `cur_cos_reaching_defs`, and its outside-definition set to be the current value of `cur_outside_reaching_defs`. (We store the cos-reaching-definition and outside-reaching-definition sets for each instruction as Deco annotations on that instruction, so that they will be accessible in constant time.) Then, for each operand `r` in the definition set of this instruction, we set `cur_reaching_defs[r]` to be a pointer to this instruction, since the definition in this instruction will override the current reaching definition. The definition of `r` in this instruction will also override any outside reaching definition, so we set `cur_outside_reaching_defs[r]` to be `FALSE`. When we reach the end of this loop, we will have set the cos-reaching-definition and outside-reaching-definition sets correctly for every instruction in the loop.

5.3 Removing Instrumentation

The first optimization we perform on the cos is to remove all remaining instrumentation code. Although we create the cos in such a way that we remove the heavyweight instrumentation point on the backedge, the cos may still contain instrumentation code. This instrumentation code can come from one of the following sources:

- A sequence of one or more instructions that increment the path register.
- If the decoder performed any partial inlining, the cos will include the heavyweight instrumentation point for the exit from the inlined procedure.
- Also, since the path register is a callee-saved register, an inlined code sequence will contain instructions intended to save the path register on the stack before executing the procedure body, and retrieve it from the stack before exiting the procedure.

Clearly, none of these residual pieces of instrumentation code is useful inside the cos; they simply waste cycles. Therefore, we have a simple optimization which recognizes such code sequences. This is very easy to do, since they are the only instructions that use the path register. Once we recognize them, we delete them from the cos.

However, removing the path register adjustments from the cos introduces a slight problem: the value of the path register when we fall out of the cos will be incorrect, because we have not been updating it properly. Our solution for this is fairly straightforward. Recall that the path register is reset by the heavyweight instrumentation point on each backedge. Since the top of the cos is a backedge target, one possible value for the path register when we enter the cos is zero.

If we assume the path register was zero when we entered the cos, it is fairly easy to compute the value it should have at each of the stubs. We simply sum the values of all the increments between the top of the cos and the conditional branch whose target is the given stub. We then insert compensation code into the stub which loads the proper immediate value into the path register before returning control flow to the original code. By assuming that we always enter the cos along the backedge, we introduce a very slight bias into our run-time profiler, but we do not consider this a particularly significant problem.

There is a similar problem with removing inlined stack saves and stack restores of the path register. If we fall off the cos in the middle of an inlined code sequence, the end of the procedure in the original code will try to load the value of the path register from the stack before returning. If we have not stored the value correctly, garbage will be loaded into the path register. Therefore, in addition to loading a proper immediate value into the path register, a stub may also contain compensation code which loads the proper immediate value into the appropriate stack locations for all the inlined code sequences in which it lies. For example, a stub may represent a fall-off point which is in a partially-inlined sequence in the middle of another partially-inlined sequence. In this case, we have to fix the appropriate stack location for both procedures.

5.4 Removing Useless Stack Operations

The path register may not be the only useless value the cos stores on the stack. Alpha calling conventions state that a called procedure must store onto the stack the value of any callee-saved register for which a definition exists somewhere in the procedure. However, since Deco inlines only a single path through a given procedure, the cos may not contain the definition which necessitated saving the register on the stack. Thus, from the point of view of the cos loop, the load and store of this stack operation is useless. We can optimize the code by removing the loads and pushing the stores to the stubs. (This is exactly what we did to the path register stores and loads in the previous section; we will not repeat the code-motion algorithm here.)

The only remaining problem is detecting useless stack operations. If we have the decoder do a slight bit of extra work, this can be done in a single iteration over the cos loop. As we decode the hot path, the partial inlining unit annotates each stack load with a pointer to the associated stack store instruction. This way, the optimizer can simply iterate over the cos loop body, and each time it sees a stack load, it can retrieve the associated stack store in constant time.

Let `x` be the register associated with a particular load-store pair (i.e. the store looks like `store $x->offset($sp)` and the load looks like `load  $x<-offset($sp)`.) If the cos reaching definition of `x` at the store is the same as the cos reaching definition of `x` at the load, we are guaranteed that the value of `x` remains unchanged between the two points. Therefore, the stack-load pair is unnecessary in the body of the cos and can be removed. This condition can be computed in constant time for each load-store pair, since it simply requires looking at the already-computed cos-reaching-definition information for each instruction.

5.5 WSS: The World's Simplest Scheduler

The goal of the optimizations we have described so far (as well as the goal of partial dead code elimination, which will be described in the next chapter) is to reduce the static instruction count of the cos loop. However, reducing the static instruction count inside the loop does not necessarily mean the loop will actually execute faster. There is a machine-specific reason for this which we have observed affecting the running time of our current implementation.

We have been running Deco on a Digital Alpha 21064. This is a dual-issue superscalar architecture, meaning that it tries to execute two machine instructions at a time [9]. A machine instruction is 32-bits long, so two instructions can fit in a 64-bit space in memory. If the current program counter is 64-bit aligned (i.e. the byte address is divisible by eight), the instruction fetch unit fetches two instructions at once. If the program counter is not 64-bit aligned, it fetches only a single instruction (so that it can return the program counter to 64-bit alignment and continue dual-fetching).

Assuming the fetch unit grabs two instructions, it looks at the two instructions to see if they can be issued in parallel. There are a number of reasons two instructions might not be able to issue simultaneously: the first defines a register which is needed by the second, the first has the potential to change the program counter (meaning that we might not want to execute the second at all), or they both require the same functional unit in the CPU. If the issuing unit determines that the two instructions can be dual-issued, they are fed into the pipeline simultaneously. If they cannot be dual-issued, the second is issued after the first.

Since two instructions are dual-issued only if they are fetched simultaneously, instruction sequence alignment is an important performance issue. If we have two instructions that could be dual-issued but the address of the first instruction is not 64-bit aligned, the execution of the instructions will take two cycles rather than one.

Removing instructions from the cos loop alters the instruction alignments and is thus not necessarily a good thing. It is possible that the speedups we expect to see from reducing the static instruction count within the cos loop will not happen because our realignments have increased the cycle count of the loop (or simply left it the same, in which case the program will still run slower because of the compensation code we have added to the stubs.)

To help alleviate this problem, we have written a very simple scheduler called WSS (which stands for World’s Simplest Scheduler) to assist the encoder in laying out machine instructions in the cos. WSS does not move instructions around, as a more aggressive scheduler would; it simply inserts `nops` in certain places in order to better align the code. (`nops` are useful in aligning code since they can be dual-issued with any instruction.)

We define a *collection* to be a maximum-length list of consecutive instructions x_1...x_n, such that for all `i <= n-1`, x_i can be dual-issued with x_{i+1}. The time to execute a collection where `n` is odd is therefore `(n+1)/2` cycles. This cycle time is independent of whether or not the beginning of the collection is 64-bit aligned: in each case, either the first or the last instruction of the collection will have to single-issue. The time to execute a collection where `n` is even is `n/2` cycles if the beginning of the collection is 64-bit aligned (since all pairs can dual-issue), but `n/2+1` cycles if the beginning of the collection is not 64-bit aligned, since the first and the last instruction in the collection will single-issue.

WSS makes sure that we never lay out an even-length collection in such a way that it takes the extra cycle. To better lay out code, WSS calculates the length of the collection we are about to write into the cos. If the collection length is even and the current address we are about to write to is not 64-bit aligned, it inserts a `nop`.

WSS is not an aggressive scheduler. Using it does not guarantee that the optimized cos will run faster than the unoptimized one. If we are only able to remove a small number of instructions, it is still quite possible that the alignment issues will outweigh the static instruction count reduction. The goal of WSS is just to fix obvious cases where removing instructions increases the cycle count.

Chapter 6 Partial Dead Code Elimination

The other main optimization currently performed by the Deco run-time optimizer is *partial dead code elimination*. This is a broad designation for an optimization that is broken up into a number of distinct phases. The goal of this optimization is to find instructions that do not need to be on the cos loop and push them to the stubs.

6.1 Partially Dead Code

An operand x is called *dead* at a point p in program code if no possible control-flow path exists from p to any instruction which uses x without x first being redefined. An instruction is considered dead if its only effect on program state is to define dead operands. Clearly, dead instructions are useless, and can be removed without altering program semantics. *Dead-code elimination* is a compile-time optimization that removes all dead instructions from a procedure's CFG [1,19,20].

However, the cos regions extracted by Deco are not complete procedure CFGs; they include only one path of control flow through the loop body. A more useful concept in this scenario is the idea of an operand being *partially dead* along a path. We say an operand x is partially dead at a point p along a path if, following the control flow of that path from point p, x is not used before it is redefined. An instruction is partially dead with respect to a particular path if its only effect on program state is to define operands which are partially dead along that path. Partial dead code elimination is an optimization we have written to detect partially dead instructions and move them from the body of the cos into the stubs. An illustration of dead and partially dead code is given in Figure 8.

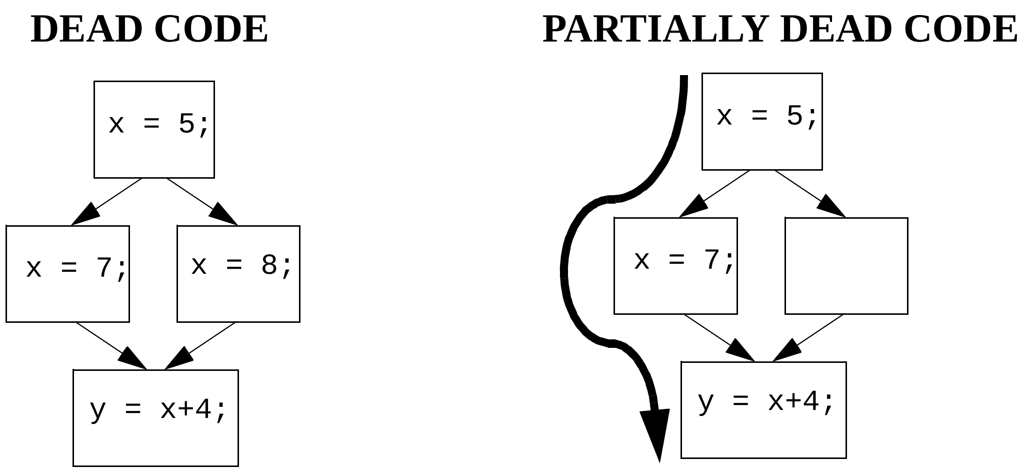

Figure 8. Dead and partially dead code. In the CFG on the left, the statement x=5 is dead. In the CFG on the right, it is partially dead along the indicated path.

6.2 Copy Propagation

In order to expose more instructions as partially dead, we perform *copy propagation* [1,19,20] on the cos. A *copy* is an instruction whose only effect on program state is to duplicate the value of one register in another register. Consider, for example, the copy statement `mov $y<-$x`. After this instruction is executed, `x` and `y` will have the same value, until we reach an instruction which redefines one of them. The idea behind copy propagation is to substitute `x` for `y` in all statements after the copy in which the values of `x` and `y` remain identical. Figure 9 shows an example of this: `x` is substituted for `y` in the instruction in the second block. Clearly, doing this will not affect program semantics.

This may, at first glance, appear to be a rather pointless thing to do. However, it has the potential to expose the copy as being partially dead, as we show in Figure 9. Suppose, for example, the copy of `x` into `y` occurs right before a conditional branch. Along one path out of the branch, `x` is redefined before y, and thus the instruction stream requires distinct registers for the values in both `x` and `y`. Along the other path out of the branch, `y` is redefined before `x`, and thus at no point did we need separate values in each register. The copy is partially dead along the second path, but not the first. By making explicit the points at which we really do need the particular value to be in both registers, copy propagation

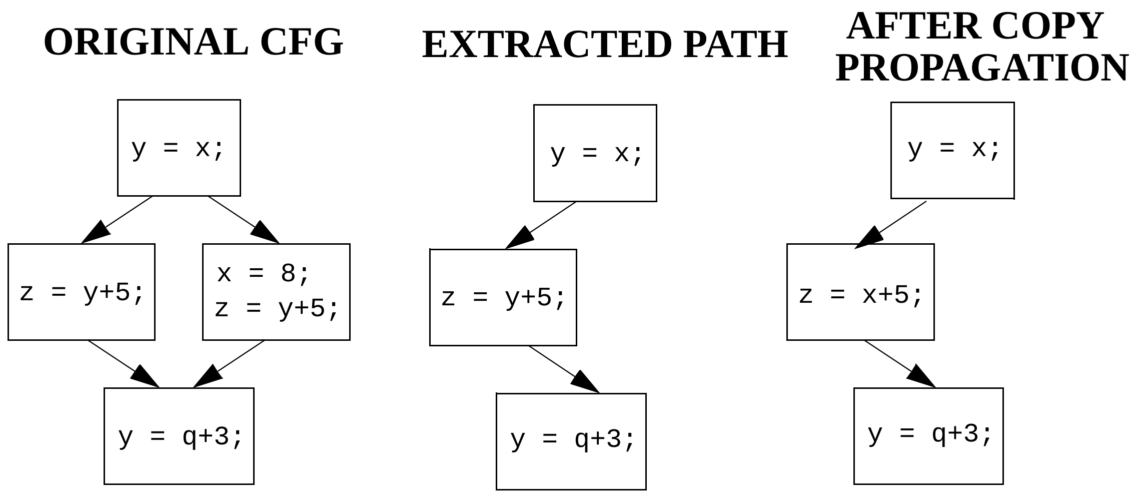

Figure 9. An example of how copy propagation on an extracted path can expose partially dead copies.

helps to expose partially dead copies. Note that the statement in the top block in Figure 9 becomes partially dead only after we perform copy propagation.

Our implementation of copy propagation uses our reaching-definition information and requires just a single iteration over the loop body. For each instruction `i` in the cos, we consider all its source registers. For each source register `y`, we query the cos-reaching-definition set of `i` to see if the cos reaching definition of `y` is a copy instruction `c` that copies the value of some register `x` into register `y`. If this is the case, we can then replace all instances of `y` in `i` with `x` if the following conditions are satisfied:

- `i` has no outside reaching definition of `y`. This condition assures that `c` is the only definition of `y` that can ever reach `i`.
- The cos reaching definition for `x` at `c` is the same as the cos reaching definition for `x` at `i`. This condition assures that `x` is not redefined between `c` and `i`.
- There is not an outside reaching definition of `x` that reaches `i` but does not reach `c`. This condition assures that the value of `x` at `i` will always be the same as the value of `x` at `c`.

Checking these conditions requires only constant-time reaching-definition lookups. The register substitutions can also be done in constant time.

6.3 Finding Partially Dead Code

Once we have performed copy propagation, our next step is to find partially dead instructions we hope to push to the stubs. Our algorithm for this is derived from Morgan's algorithm for dead code elimination [19]. He takes a backwards approach to finding dead code; rather than trying to identify dead instructions explicitly, he instead identifies *necessary* instructions. An instruction is considered necessary if it makes useful changes to program state. Any statement which is not necessary must be dead and can be removed.

Morgan marks all instructions in a procedure that should not be deleted; our algorithm marks all instructions in a cos that should not be moved to the stubs. We distinguish necessary instructions by marking them with a `NECESSARY` annotation (so that we can look up whether a given instruction is necessary in constant time). We begin by performing a single pass over the cos, initially marking as necessary instructions of the following types:

- Call instructions. Such instructions are "black boxes" which could potentially change program state in important ways.
- Branches and jumps. These instructions can change the program counter and cause us to fall out of the cos.
- Instructions that write to memory. Since we do not dynamically track different memory locations, we must consider all writes to memory as having a necessary effect on program state.

As we mark these instructions necessary, we add them to a queue called the *worklist*. Once we have completed this first pass, we begin popping instructions off the worklist. For each instruction `i` we pop off the worklist, we look at its source registers. Since `i` is necessary, the instructions that define its sources must also be necessary. Therefore, for each source of `i`, we mark its cos reaching definition as necessary and add it to the worklist if this cos reaching definition has not already been marked as necessary. This process continues until the worklist is empty, at which point we will have identified all instructions that are necessary for the loop body.

However, this is not quite sufficient. Just because the result of an instruction is not used in the cos loop does not mean it is partially dead and can be moved to a stub. Consider, for example, a loop in which the instruction `add $x<-$x,1` is the only instruction

```
srcs(i): the source registers of instruction i
reaching_defs(srcs(i)): the set of instructions that are
   reaching definitions for source registers of i

/* First pass: initialize instructions as necessary */
for(i = first_instruction to last_instruction)
   if(writes_memory(i) || is_call(i) || modifies_pc(i))
   {
      mark_as_necessary(i);
      worklist->append(i);
   }

/* Worklist algorithm to find all remaining necessary
   instructions */
while(!worklist->is_empty())
{
   necessary_instr = worklist->pop();
   for(rd in reaching_defs(srcs(i))
   {
      if(!marked_as_necessary(rd))
      {
         mark_as_necessary(rd);
         worklist->append(rd);
      }
   }
}

/* Iterate over the cos to find maybe_necessary instructions */
for(i = first_instruction to last_instruction)
{
   if(!marked_as_necessary(i))
   {
      for(rd in reaching_defs(srcs(i))
      {
         if(!marked_as_necessary(rd) &&
             !marked_as_maybe_necessary(rd))
            mark_as_maybe_necessary(rd);
      }
   }
}
```

Figure 10. Pseudocode for identifying all necessary and maybe-necessary instructions in the cos.

which uses or defines the register x. The above algorithm would not identify it as necessary, since none of the other necessary instructions in the loop use its result. But clearly this instruction needs to remain in the cos loop rather than being pushed to the stubs, because the exact value in x depends on how many times we execute the loop.

We therefore introduce the concept of *maybe-necessary*. A maybe-necessary instruction is a non-necessary instruction which computes a value used by some other non-necessary instruction. To find maybe-necessary instructions, we simply iterate over all the instructions in the loop that are not marked necessary, and mark the reaching definitions of their sources with a `maybe-necessary` annotation. Any instruction which is neither necessary nor maybe-necessary is partially dead, since no instruction in the loop uses its result.

Pseudocode for the algorithm to find necessary and maybe-necessary instructions appears in Figure 10. Note that the algorithm runs in linear time. Our initial necessary-marking pass runs over the cos exactly once. Since we never add an instruction to the worklist twice, the maximum number of times we can pop an instruction off the worklist is equal to the number of instructions in the cos loop. The only operations we perform when popping an instruction off the worklist are constant-time reaching-definition lookups. Finally, our maybe-necessary marking pass makes only a single iteration over the cos.

One might wonder why we go to the trouble of distinguishing between necessary and maybe-necessary. We could create an equivalent algorithm by condensing our current algorithm into a single loop over the cos that marks the reaching definition of every source of every instruction as "not partially dead".

The reason we make the distinction between necessary and maybe-necessary is that we plan to perform multiple iterations of dead-code elimination. In order to do this, we need to recompute which instructions are partially dead after each iteration, since the cos has changed due to the removal of the partially dead code we found in the last iteration. Clearly the set of necessary instructions remains the same for all iterations: we never remove necessary instructions, so membership in the set is static. But membership in the maybe-necessary set can change, as in the following situation:

```
loop:
   1: add $a <- $a, 1
   2: add $b <- $c, 1
   3: add $d <- $b, 1
   4: cmpgt $e <- $f, 0
   5: beq $e, loop
```

Instruction 5 is necessary because it is a branch. Instruction 4 is necessary because it computes a value used in instruction 5. Instruction 3 is neither necessary nor maybe-necessary. Instruction 2 is maybe-necessary, because it computes the value of `b` used by instruction 3. Instruction 1 is maybe-necessary, since it is its own reaching definition of `a`. The first iteration will remove instruction 3, leaving us with the following loop:

```
loop:
   1: add $a <- $a, 1
   2: add $b <- $c, 1
   4: cmpgt $e <- $f, 0
   5: beq $e, loop
```

Instructions 4 and 5 are obviously still necessary, and instruction 1 is still maybe-necessary. Note, however, that instruction 2 is no longer maybe-necessary, so the next iteration will remove it.

Because we make a distinction between necessary and maybe-necessary instructions, subsequent iterations require only that we recompute the maybe-necessary set. This does not take as long as computing all the reaching definition dependencies of all the instructions in the loop.

6.4 Removing Partially Dead Code

Once we have figured out which instructions are partially dead, we would like to remove them from the cos loop. However, we cannot completely eliminate them, because instructions outside the cos may depend on their having been executed at least once. Therefore, we may need to add the instruction as compensation code in one or more of the stubs.

Suppose we find that an instruction `i`, with destination `r,` is partially dead. We need to move `i` into all stubs `s` for which the following two conditions hold:

- `i` is the cos reaching definition for `r` at the top of `s` (i.e. `i` is the cos reaching definition for `r` at the conditional branch whose target is the stub.) We should move `i` into `s` only if there is no redefinition of `r` between `i` and the conditional branch whose target is `s`.
- `r` is live into the stub (i.e. there exists some control flow path from the target of the stub along which `r` is used before it is defined.) We should move `i` into `s` only if

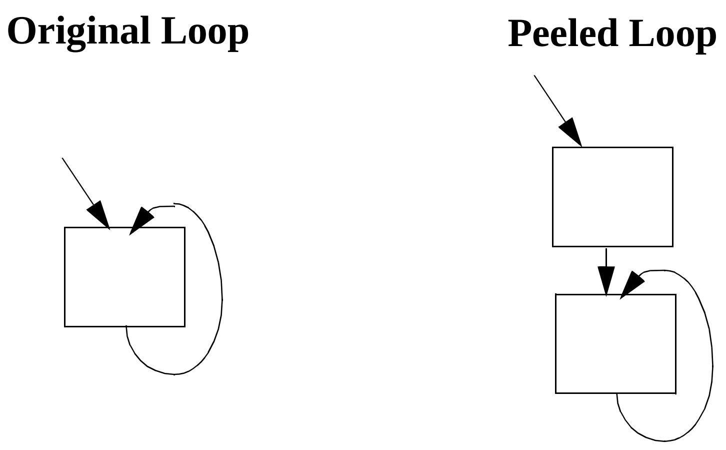

Figure 11. Loop Peeling

some control flow path out of `s` actually needs `i`'s definition of `r`. Recall that we obtain stub liveness information from the hint-passing data structure.

However, the question of whether it is actually possible to move `i` into `s` is a different issue entirely. We can move `i` into `s` only if the following conditions are true:

- `s` does not have an outside reaching definition for `r`. If `s` has an outside reaching definition for `r`, it is not necessarily correct to execute `i` inside of `s`, since we may have gotten to `s` along a path that would not normally have executed `i`.
- The cos reaching definitions for the sources of `i` are the same at `s` as they were at the original location of `i`. If this condition is not true, when we execute `i` inside of `s`, we will use incorrect sources and may potentially calculate an incorrect value for `r`.
- Similarly, none of the sources of `i` can have an outside reaching definition at `s` if it did not have one at the original location of `i`.

Our algorithm for removing partially dead code follows quite naturally from these conditions. For each partially dead instruction `i`, we compute the set of stubs into which we need to move it to preserve program correctness. This takes linear time relative to the total number of stubs. For each stub `s` in the set, we can check in constant time if `i` can be moved into `s`. If `i` can be moved into all the stubs in the set, we do so and remove `i` from the cos loop. If there exists some `s` into which we cannot move `i`, we do nothing, because it is impossible for us to legally remove `i` from the cos loop.

It is theoretically possible for us to remove the conditions having to do with outside reaching definitions. Note that the only purpose of the outside definition set is to allow us to distinguish the first iteration of the loop from subsequent iterations (since the outside definitions will be overridden after a single loop iteration if there is a reaching definition inside the loop.) Therefore, it might be advantageous for us to restructure the cos by *peeling* it. Peeling restructures a loop by making a copy of the loop body and placing it as the copy as a preheader to the loop (Figure 11 gives an example). If we were to peel the cos and then perform dead code elimination on the loop body, we would not have to worry about outside definitions, since we would be assured of having executed at least one iteration of the loop. The downside of peeling is that it increases code size. The current implementation does not do peeling, but this would be a very easy feature to add.

Chapter 7 Experimental Evaluation

This chapter gives results from the current Deco prototype. Our focus is on drawing some preliminary conclusions about the overall performance of the system; we have not attempted to parameterize Deco for optimal results.

7.1 Benchmarks

The current benchmark set for Deco consists of the following programs:

benchmark	description	data set
wc	UNIX word count program	etext of *My Bondage and My Freedom* by Fredrick Douglass
compress	Lempel/Ziv file compression utility (SPECint 92)	etext of *My Bondage and My Freedom* by Fredrick Douglass
eqntott	translates boolean equations to truth values (SPECint 92)	fixed to floating point encoder
li	XLISP interpreter (SPECint 95)	SPECint 95 train input
m88ksim	microprocessor simulator (SPECint 95)	Dhrystone benchmark
rend	a.k.a rayshade, raytracer developed by Andersen for experiments with run-time partial evaluation [2]	rend scene 1

Table 1: Deco benchmark suite

This benchmark set was chosen with an eye towards programs we thought might benefit most from dynamic optimization. First of all, we attempted to select highly input-dependent applications. All of the above benchmarks meet this requirement; `li`, `m88ksim`, and `rend` have a particularly wide range of possible inputs.

Secondly, we had a preference for benchmarks that spend a great deal of their execution time in code which appears acyclic from an intraprocedural standpoint. As we discussed briefly in Chapter 3, if an application exhibits such behavior, it is probably an indication of the existence of one or more heavily-executed loops that make calls into acyclic procedures. Such loops would be prime candidates for partial inlining of the calls to acyclic procedures, and might therefore exhibit large performance benefits. Hank et al. [16] suggest that `li` and `eqntott` have this property, and `rend` also has a very high percentage of acyclic code.

7.2 Methodology

We ran all of our experiments on an Alpha 21064 [9], running Digital UNIX 4.0 in single-user mode.

The Deco parameters were set as follows:

- The interval length for the trip mechanism was 1000 executions of backedge-terminated paths per procedure.
- The trip threshold for a backedge-terminated path is 50%. That is, if a backedge-terminated path executed with greater than 50% frequency during a particular interval, we selected that path for optimization.
- The trip threshold for partial inlining was 50%. If the procedure corresponding to a call site being decoded had an entry-exit path which executed more than 50% of the time (relative to the set of all entry-exit paths), the call site was partially inlined with that entry-exit path.
- The cos cache contained five entries.

Running times for programs were computed using cycle counts, as recorded by the Alpha processor cycle counter (PCC). Since the PCC is only 32 bits long, it overflowed frequently. Therefore, a small amount of code was inserted to cause timer interrupts every four seconds, allowing us to sample the counter at intervals smaller than the time required for it to overflow. These timer interrupts had a negligible effect on the overall running time. In order to adjust for variance between runs that might affect the timing (paging, for instance), we took the minimum running time from a set of several program runs.

7.3 Deco Overhead

Our first series of experiments attempts to determine the amount of overhead introduced by Deco. To do this, we compare the running time of each of our benchmarks compiled under SUIF 1.1.2 (inserting only the cycle-counting instrumentation) with the running time for that benchmark compiled under Deco.

We expect our overhead to come from two sources: profiling costs (instrumentation, the run-time profiler, and the trip mechanism) and optimizing costs (the decoder, the optimizer, and the encoder). In order to separate these costs, we ran our Deco-ized executables with two versions of the Deco shared libraries. In the first set of runs, we used a version of the libraries that identifies paths and trips on them, but does not decode or optimize them. In the second set of runs, we used a version of the libraries that identifies paths, trips on them, decodes them, optimizes them, and writes them into a cos, but does not patch the optimized cos back into the original code. The first set of runs therefore incurs only profiling costs, the second set incurs both profiling costs and optimizing costs. Since neither version of the libraries includes the patcher, neither set of runs receives any benefit from Deco; this allows us to view only the overhead.

Figure 12 shows the overhead from each part of the Deco system. The cumulative overhead is quite high in all of our benchmarks, ranging from 236% for `compress` to 615% for `m88ksim`. This high level of overhead is not unexpected. Neither the profiler nor the optimizer has been implemented with efficiency as the primary goal. In particular, the profiler incurs a great deal of procedure call overhead: as many as three procedure calls may be necessary to update the profile table and query the trip mechanism. When we consider that this cost is incurred on the execution of every backedge and at every return point, it is quite understandable that the profile overhead is so high.

The overhead from the optimizer is actually quite low relative to the overhead of the profiler. The two exceptions to this are `eqntott` and `li`. The reasons for this will become clearer once we examine some statistics about the paths pulled out by Deco.

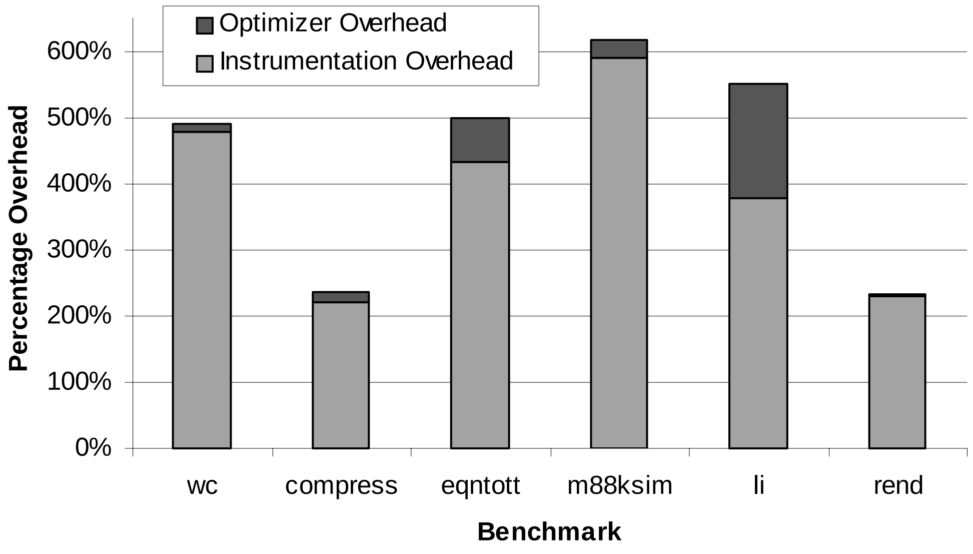

Figure 12. The overhead in a Deco-ized executable.

7.4 Path Extraction

Table 2 gives information about the paths Deco extracts. The first column gives information about how many total times the trip mechanism passed a path to the run-time optimizer. The second column tells us how many of these paths were unique. A smaller number in column two than in column one implies that we are pulling out the same paths multiple times during the program run (i.e. they trip again after being evicted from the cos cache).

benchmark	number of extracted paths	number of unique extracted paths	number of unique procedures
wc	1	1	1
compress	11	5	3
eqntott	958	25	18
m8kksim	33	15	14
li	73	12	10
rend	1	1	1

Table 2: Paths extracted by Deco during benchmark runs.

The third column shows how many unique procedures are represented by these unique paths. Recall that the current system cannot have more than one cos per procedure at any given time; a smaller number in column three than in column two is evidence of competition for this one cache block. This competition may be due to unwanted thrashing, if one procedure has two equally hot paths which continuously fight each other for space in the cache. However, the competition may simply be due to the phased nature of the program's execution: because one path in a procedure is hot for a while, then another path becomes hot. In our tests, we have seen both thrashing and phased behavior.

As we would expect, the optimizer overhead correlates with the total number of paths pulled out over the course of the run. The next question to ask is why some benchmarks pull out a much larger number of paths than others. One might think that a longer-running application would pull out more paths, and thus number of paths would be proportional to the length of the run. However, this is not the case: `rend`, for instance, is by far the longest-running of our benchmarks, yet it extracts only a single path. The interval length for our trip mechanism is set low enough that there are many interrupts during the course of even a short program run; a small number of extracted paths therefore implies that there were few paths which met our "hotness" criteria. This is a good thing: we want the number of paths we extract to reflect how many hot paths there were, rather than reflecting how long the program ran.

Another explanation might be that a cos cache size of five is too small, and that the random replacement policy is evicting paths which are immediately tripping again. To test this theory, we ran the system with larger cache sizes. However, the total number of paths pulled out did not decrease significantly with a larger cos cache.

From looking at traces of the exact paths that were pulled out, it appears that the main reason some benchmarks pull out more paths than others is due to thrashing between paths in the same procedure. The traces for both `eqntott` and `li` exhibit long sequences of thrashing behavior. This shows us that having only one cos per procedure at any given time can be quite limiting when our coses consist of single Ball-Larus paths.

One possible solution to this problem would be to allow more than one cos per procedure in the cache. However, there is a ceiling on how much this will improve the situation, since our current prototype cannot simultaneously patch two coses with the same patch point. A longer-term solution is to form more complex regions that represent the union of thrashing Ball-Larus paths within the same procedure.

7.5 Evaluating the Trip Mechanism

Now that we understand why the trip mechanism pulls out as many paths as it does, the next logical question to ask is whether it is pulling out useful paths. In other words, we would like to know what percentage of our execution time is spent in the paths in the cos cache.

A simple way to determine this comes from using a version of the Deco shared libraries that patches in minimally-optimized coses. Recall that pulling out a path into a cos removes all instrumentation on the backedge. We can remove the remaining instrumentation code from the cos using the optimization described in Section 5.3. Note that the only difference between the cos and the original code is that the cos lacks all instrumentation.

We can therefore get a measure of how often we are executing the code in our coses by examining how much the profiling overhead is reduced in this version from the version that does full profiling and optimizing but no patching (the second set of Deco runs we did in Section 7.3.) The more we execute in the uninstrumented coses, the lower our profiling overhead will be.

Figure 13 shows the results of this experiment. The solid bars indicate what percentage of the profiling overhead remains when we patch in instrumentation-free coses. The results of this experiment indicate that despite the small size of our cos cache (five entries), we get a great deal of dynamic path coverage. Only `rend` removes less than 35% of the profiling overhead; `wc`, `eqntott`, and `m88ksim` all remove over 70%. This implies that our trip mechanism is selecting reasonably hot paths, and that our cos cache is doing a fairly good job of keeping them around.

The benchmark `rend` exhibits an uncharacteristically low overhead reduction. `rend` differs from the other applications in our benchmark suite in that most of its code is located in small helper functions. This means that an usually high percentage of our instrumentation overhead is coming from executing heavyweight instrumentation points at

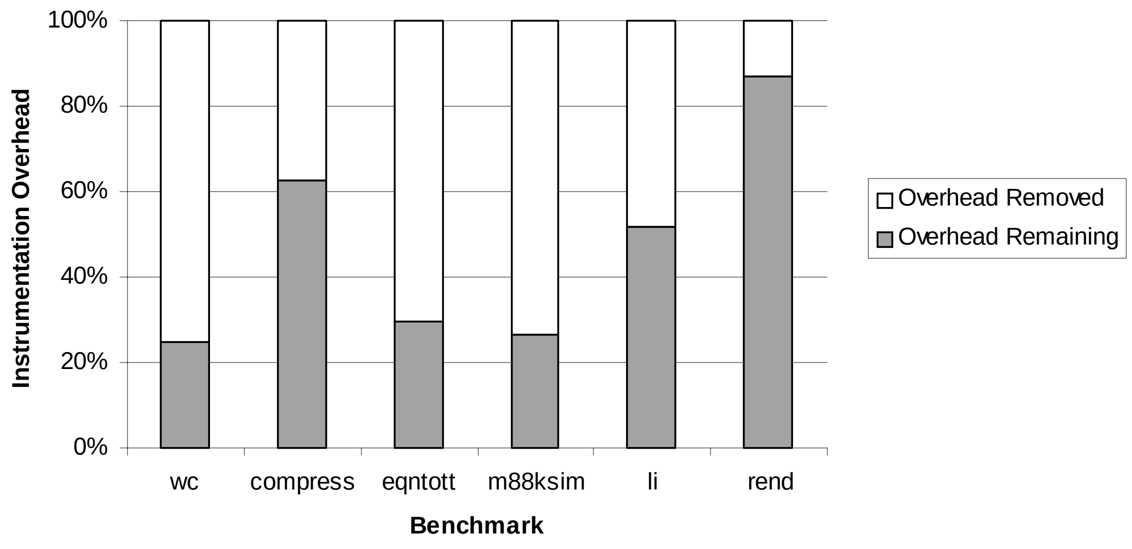

Figure 13. Overhead reduction due to execution of uninstrumented coses.

function exits, rather than on loop backedges. There is only one hot loop path, which gets pulled out very early in program execution. Although the decoder partially inlines five of the calls along this path, this is only a small fraction of our profiling overhead.

We have also observed that we are often staying in the same cos for many consecutive executions in a row. We ran an experiment where we did not remove instrumentation from the cos; in this experiment the only profiling reduction came from the removal of instrumentation along the backedge. Therefore, in order for us to observe a great deal of overhead reduction, we would have to execute the backedges of the coses many times. Running this experiment gave us comparable performance enhancements to the ones shown in Figure 13. This implies that not only are we executing in coses frequently, but we are staying in coses for extended periods of time.

7.6 Static Instruction Removal

Finally, we would like to determine the effectiveness of our current optimization suite. Since the intended effect of all of our optimizations (except WSS) is to remove instructions, one way to measure their effectiveness is to see how much they reduce the instruction count within our coses.

We ran a version of the Deco shared libraries which implemented our entire optimization suite, and collected information about how much these optimizations shortened the lengths of our paths. Table 3 summarizes these results.

benchmark	number of extracted paths	average number of partial inlines	average path length		percent reduction
			before optimization	after optimization	
wc	1	0	26	25	3.8%
compress	11	0	24.36	23	5.5%
eqntott	958	0.0031	18.3	15.9	13.0%
m88ksim	33	0.39	39.12	34.84	11.0%
li	73	0.82	58.2	49.4	15.1%
rend	1	5	205	184	10.2%

Table 3: The effect of the Deco optimization suite on the lengths of extracted paths.

Column one shows the number of paths we pull out over the course of the program run. Column two tells us how many calls, on average, we were able to partially inline for each path; the number presented is the total number of calls the decoder was able to partially inline divided by the total number of paths pulled out (the number in column one). Column three shows the average length of a path before optimization, including partial inlining, but not including any instrumentation code; column four shows the average length of the paths from column three after we optimize them. In column five, we see the percentage reduction in path length between column three and column four.

These results show us that the current Deco optimizations are, on average, removing around 10% of the static instructions from our cos loop. This is somewhat higher than the reductions reported for hot-cold-optimization [7]. Partial inlining seems to be of great assistance in achieving these path length reductions; it exposes useless interprocedural code which we can then remove. Note that the path length reductions for the benchmarks which perform partial inlining are much higher than for the benchmarks that do none.

7.7 Dynamic Benefit of Optimizations

The dynamic benefit of optimization is a function both of how much we optimize paths and how often we execute the optimized paths. The previous two sections have

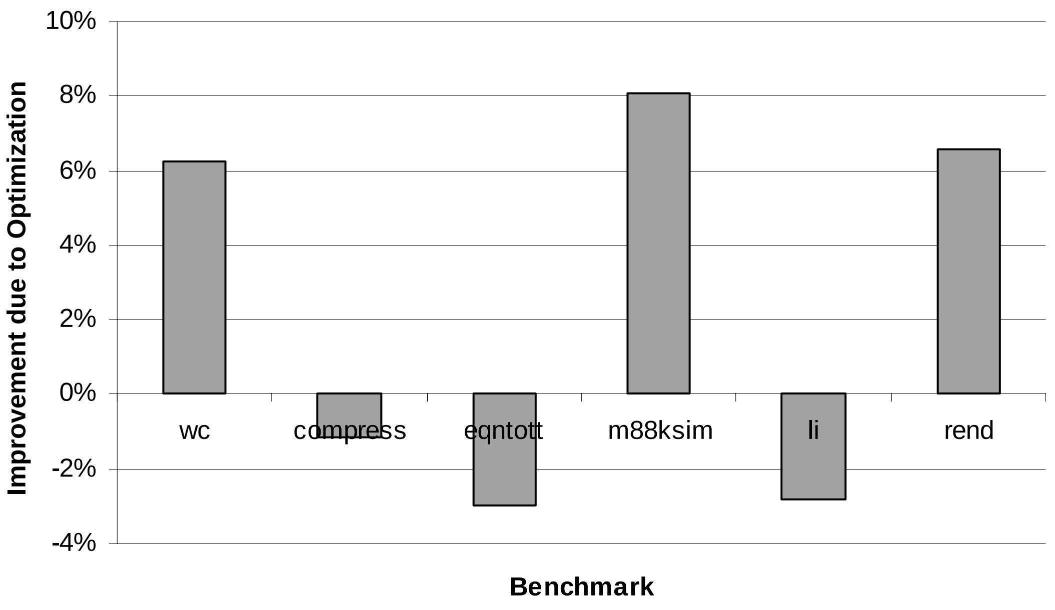

Figure 14. Improvement in running time due to optimizations.

shown that we pick good coses and that we are able to remove instructions from the coses we select. This section attempts to put these two elements together and discuss how much our optimizations are able to reduce program running time.

To determine the effectiveness of our optimizations, we ran Deco with all optimizations and partial inlining turned on and compared the running time to a version of Deco which only removed instrumentation, and did not perform partial inlining. The results of this experiment are shown in Figure 14.

The first thing one notices about this graph is that the running time does not always improve when we run our optimizations. We will discuss each of the benchmarks in turn and explain why they did or did not perform better under our current optimization suite.

`wc`, `m88ksim`, and `rend` all improve significantly with our optimizations. This is what we would expect to see, since we are removing instructions from frequently executed paths. `wc` actually performs much better than we might have predicted: recall that Deco removes just one instruction from its hot path. The reason for this is that WSS is able to lay out the code along the hot path in a more optimal way than was possible statically, saving an extra cycle in the execution of the hot path. The dynamic effect of doing this reduces the execution time even more than the instruction-removal statistics would indicate.

However, just as scheduling issues can yield benefits, they can also cause problems. This is what we see in both `compress` and `eqntott`. Table 3 shows that we only remove 1.36 instructions per path from `compress` and 2.4 instructions per path from `eqntott`. Recall from Section 5.5 that when we remove such a small number of instructions, we may not actually decrease the cycle count of the cos (even though we run WSS). Furthermore, removing these instructions increases the cycle count of the stubs by adding compensation code. Therefore, the net effect of the instruction removal is a greater dynamic cycle count. This is what appears to be happening in both `compress` and `eqntott`: we are pulling out short loops and not removing very many instructions from them. More aggressive optimizations will probably be necessary before we see any benefit from either.

The performance loss in `li` is more difficult to explain. Figure 13 shows a decent reduction in instrumentation overhead for `li`, implying that we are choosing frequently-executed paths. Furthermore, Table 3 indicates that in `li` we reduce the average length of a hot path by 8.6 instructions. The combination of these results suggests that `li` would exhibit improved performance; it is therefore surprising to observe that optimizations caused it to perform worse.

We believe that the reason `li` performs poorly is that we do not stay in the cos for very long each time we enter it. In order to see performance improvement, we must execute the cos backedge enough times that the benefit from shortening the cos loop outweighs the cost of lengthening the stubs. If we do not stay on the cos loop for very long each time we enter the cos, our optimizations will actually cause a performance decrease.

We earlier discussed how we have observed thrashing behavior in `li`. There is one procedure which seems to have two equally hot paths; this accounts for almost all of the thrashing. This thrashing behavior may be an indication that we do not have very many repeated executions of one path before we see an execution of the other—this would imply that we fall off the cos after very few iterations of the backedge. Therefore, the large number of instructions we move from the cos loop into the stubs would wind up hurting performance. The behavior of the `li` benchmark provides further motivation for pulling out regions which are not simply single Ball-Larus paths, in order to avoid this type of detrimental thrashing behavior in procedures with more than one hot path.

7.8 Summary

We have examined the performance impact of Deco's instrumentation code, run-time profiler, and run-time optimizer. Our analysis shows the current status of the system and in what areas we need to improve. We are doing better than we might have expected in some areas, such as the selection of good paths. However, the results also indicate the need for more complex regions and more aggressive optimizations, so that scheduling issues and thrashing do not continue to plague the performance of the optimizer.

Chapter 8 Related Work

Specialized run-time techniques have been successfully applied to a number of different areas. For example, one application has been to optimize dynamic typing in the object-oriented language SELF, in order to eliminate some of the overhead of virtual table look-ups [6,18]. This method optimizes frequently-executed sections of code by replacing dynamically-dispatched method calls with inlined code from the most commonly-observed dynamic call targets. The focus of the SELF project was on object-oriented language-specific optimizations, which differs greatly from our goal of machine-specific optimization.

Engler et al. have attempted to systematize the problem of dynamic code generation. They have developed ‘C (Tick-C), a dialect of C that allows the programmer to specify certain sections of code whose compilation will be delayed until run-time [11,21,22]. The generated code can then be optimized to take advantage of run-time information such as *run-time constants* (program parameters such as command-line arguments, which are unknown at compile-time but remain constant throughout the run of the program) and *dynamic partial evaluation* (specializing functions with respect to certain arguments). An example of dynamic partial evaluation would be the specialization of an exponentiation function for a certain exponent. Grant et al. have a similar system called DyC, in which the programmer identifies variables, rather than code regions, for potential run-time optimization [14]. Related to the work on ‘C has been the development of fast dynamic code generators to efficiently perform the necessary run-time compilation [10,12].

Unlike the work presented in this thesis, these methods place the burden on the programmer to identify regions amenable to run-time optimization. Deco uses the runtime profiler to drive input-specific optimizations, rather than depending on the programmer's knowledge to provide a summary of the application's dynamic behavior.

The increasing popularity of Java has created an interest in making Java virtual machines run faster, and some run-time techniques have been applied to this problem. Just-in-time compilers (JITs) translate portable bytecodes at run-time into native machine code, improving performance by not requiring reinterpretation of the bytecodes on subsequent executions of the compiled code sequence. More advanced JIT compilers, such as Sun Microsystems's HotSpot Java system [17], claim to use run-time profiles both to select regions of code as compilation candidates and to drive optimizations such as inlining and dynamic dispatch elimination. However, since these are commercial applications, detailed information about their implementation is unavailable.

Chapter 9 Future Work

Research on dynamic optimization has barely begun. The work in this thesis identifies the key issues involved in the development of a dynamic optimization system and describes a working prototype. Though the prototype is a major step in the development of a dynamic optimization system, it is just the first of many such steps. The ultimate goal should be to make Deco not simply an experimental platform, but a viable run-time optimization system that demonstrates clear performance improvements. This chapter outlines directions for future work towards this goal, with a focus on potential additions to Deco.

9.1 Lowering Instrumentation Overhead

As Chapter 7 showed, the main inefficiency of Deco today is the high amount of instrumentation overhead required for the run-time profiler. Clearly, instrumenting every path in every function throughout the lifetime of the program run is not the way to go.

Nor should this really be necessary. It is likely that many functions in a program execute so infrequently that we will never find hot paths in them; keeping statistics about such functions is therefore useless overhead. Furthermore, it is likely that not all functions exhibit phased behavior. If a function's behavior seems consistent over a long period of time, we may want to stop collecting information about it. In other words, we want to be able to adjust our instrumentation scheme dynamically so that we are only gathering information about the parts of the program that seem particularly interesting.

One way to do this is to move the insertion of instrumentation from compile-time to run-time. Initially, we would instrument none of the program code. Our initial feedback information during the program run would come from *sampling*. Sampling is a technique which uses periodic cycle counter interrupts to determine which instructions are being executed frequently.

While sampling does not give us nearly as much information as path profiling, it does give us a general sense of which regions in the program may be hot. The big advantage of sampling is that it gives us this information with extremely low overhead. DCPI, a sampling tool for Alpha microprocessors, has an overhead of only 0.5%-3.0% on most applications [8].

Deco can use the run-time information provided by DCPI or another sampling tool to drive the run-time insertion and removal of instrumentation code. When sampling finds a hot procedure, we add instrumentation code to it; this will provide the run-time profiler with the detailed information it needs to pull out hot regions. If sampling tells us that an instrumented procedure is no longer hot, or the observed behavior of the procedure is relatively stable, we remove the instrumentation. Since we would be instrumenting a relatively small number of procedures at any given time, it would be realistic to convert the profile table into an array, rather than the current data structure. This should reduce the overhead even further.

Driving our profiling by means of sampling would also address another weakness of the current Deco system: the fact that we only instrument procedures available at compile-time. By performing instrumentation completely at run time, we would be able to profile any executed procedure.

Current research efforts at Harvard are looking into the problems of inserting and removing instrumentation code at run time. Hopefully, with these additions, the overhead of Deco instrumentation can be drastically reduced.

9.2 Lowering Optimization Overhead

Many of the benchmarks have a sizable proportion of overhead associated with pulling out and optimizing paths. Although not nearly as high as the instrumentation costs, this overhead is still important. Because of the emphasis on extensibility and modularity, very little effort has been made to optimize the Deco IR. It makes use of C++ classes and virtual functions, which are not particularly efficient. We are fairly confident that a significant improvement will be seen by re-engineering the implementation of the system with a greater emphasis on speed.

9.3 Improving the Trip Mechanism

Although our current trip mechanism performs quite well today for most of our benchmarks, there is obviously much further research to be done in this area. The current trip mechanism is a modular section of the overall system—therefore, testing out new trip mechanisms in Deco is a very simple task.

Right now, we only trip on single Ball-Larus paths. However, as we showed in Chapter 2, single Ball-Larus paths are not the only patterns that occur in program execution. It would be nice to detect superpaths (recall that a superpath is a sequence of paths which repeats often). Single Ball-Larus paths could be considered length-one superpaths.

If we view the path trace of program execution as a string, the problem of finding superpaths reduces to the problem of detecting substring periodicity. Since we need to do this as the trace is being built, an ideal algorithm for periodicity detection would perform minimal incremental computations as each new path is recorded. This would prevent the necessity of performing computations on a huge dataset.

It is not realistic to keep around the entire trace of program execution; we would run out of memory were we to try to do so. Nor should we wish to do so; older parts of the trace become less relevant as the program run moves forward. All of this implies that our algorithm might want to use some sort of sliding window protocol or aging system, methods that would get rid of old data and put greater weight on the more recent parts of the trace.

We might also wish to modify our trip mechanism so that it takes into account not only path (or superpath) frequency, but also expected benefit from optimization. As we saw in Chapter 7, longer paths showed much greater dynamic improvement from our optimization suite. In some of our benchmarks, we wasted a great deal of time identifying and extracting paths that we could not effectively optimize. Therefore, it would probably be beneficial to weight the execution counts of the trip mechanism according to certain characteristics of the region (length, type of instructions, etc.) which are conducive to optimization. That way, the trip mechanism would not simply find the most frequently executed regions; it would find the regions which, if optimized, should improve running time the most.

9.4 Better Optimizations

Our current optimization set is not ideal. Although our optimizations are fast and reduce the static instruction count, they are not always effective at lowering overall running time. Therefore, a clear area of research is the investigation of what optimizations actually do reduce running time, and on what types of regions these optimizations are most effective.

The Deco IR attempts to make it easy to add new optimizations. Especially convenient is the fact that it is compatible with the IR for Machine SUIF 2.0, making optimizations portable between the two systems. Developing optimizations in SUIF and later porting them to Deco can alleviate some of the difficulties encountered in debugging dynamically optimized code. Therefore, practically any compile-time code optimization can be implemented in Deco.

Our current optimization suite is not very aggressive—there is much room for improvement. For instance, performing full register reallocation will allow us to give precedence to the registers used on the hot path, and spill other registers whenever possible. Traub et al. give an algorithm for effective register allocation in linear time [26]; hopefully the techniques they present could be applied in a run-time reallocation pass.

Since the paths we pull out in the current implementation are superblocks, another effective optimization might be a combination of *loop unrolling* and s*uperblock scheduling*. Loop unrolling makes several copies of a loop body and lays them out consecutively, with the goal of providing the scheduler with a longer instruction stream. Superblock scheduling rearranges the order of instructions to better match the resources of the microprocessor, allowing the CPU to issue more instructions in parallel. Young shows significant benefits from compile-time path-profile-driven loop unrolling and instruction scheduling [27]; the addition of run-time profile information could potentially improve this technique significantly.

Run-time optimizers should also attempt to move beyond traditional compile-time techniques and attempt to perform optimizations that can only be performed at run time. Run-time constant propagation and dynamic partial evaluation are examples of such techniques. Engler et al. have shown a great deal of dynamic speedup with these optimizations in their ‘C system [11,21,22]; the use of run-time profiling, rather than programmer guidance, to direct them may improve their performance even further.

9.5 Cos Management

The selection of multiple coses and the idea of a cos cache highlight the need for better understanding of cos management. There are a lot of parameters which have yet to be fully explored. What is a good size for the cos cache? What is a good cache replacement policy? Effective answers to these questions will assure that we are obtaining maximal cumulative benefit from our optimized regions.

Chapter 10 Conclusion

Code optimization should be a continuous process, rather than a one-shot operation. In order for applications to run as fast as possible, optimizers should take advantage of all relevant information, whenever it becomes available. Input-specific and phase-specific profiling information only becomes available during the program run; dynamic optimization techniques are therefore an important part of the code optimization process.

This thesis has shown that dynamic optimization has the potential to be a realistic and profitable technique for achieving program speedup. Through the development of Deco, we have identified the major issues involved with building a dynamic optimization system. The implementation of our prototype offers a number of methods for addressing these different issues.

We have also presented results from our prototype and evaluated the effectiveness of the major parts of the system. Using these results as a motivation, we have suggested a number of ways in which one might extend the work presented here. It is our hope that the Deco system can provide the framework for future exploration of the dynamic optimization problem space, giving researchers the ability to answer many of the questions this thesis has raised.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishers, Reading, MA, 1988.

[2] P. H. Andersen, “Partial Evaluation Applied to Ray Tracing,” January, 1995.

[3] T. Ball and J. Larus, “Efficient Path Profiling,” *Proc. 29th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, December, 1996.

[4] T. Ball and J. Larus, “Programs Follow Paths,” *Technical Report MSR-TR-99-01*, January, 1999.

[5] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, A. Seznec. *GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications*, Technical Report PI-1153, IRISA, 1997.

[6] C. Chambers, D. Ungar, and E. Lee, “An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Langauge Based on Prototypes,” *OOPSLA ‘89 Conference Proceedings* (*SIGPLAN Notices*, 25, 10(1989), pp. 49-90).

[7] R. Cohn and P. G. Lowney, “Hot Cold Optimization of Windows/NT Applications,” *Proc. 29th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 80-89, December, 1996.

[8] J. Dean, C. Waldspurger, and W. Weihl, “Transparent, Low-Overhead Profiling on Modern Processors,” *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[9] Digital Equipment Corporation, *DECchip™ 21064-AA Microprocessor Hardware Reference Manual*, 1992.

[10] D. Engler, “VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System,” *PLDI*, 1996.

[11] D. Engler, W. Hsieh and M. F. Kaashoek, “`C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation,” *POPL*, 1996.

[12] D. Engler and T. Proebsting, “DCG: An Efficient, Retargetable Dynamic Code Generator,” *ASPLOS*, 1994.

[13] Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. “Procedure Placement using Temporal Ordering Information,” *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 303-313, December 1997.

[14] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. “An Evaluation of Staged Run-Time Optimizations in DyC,” To be published in *Proceedings of the SIGPLAN ‘99 Conference of Programming Language Design and Implementation*, 1999.

[15] R. Gupta, D. A. Berson, and J. Z. Fang, “Path Profile Guided Partial Dead Code Elimination Using Predication,” in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*. San Francisco, CA, 1997.

[16] R. Hank, W. Hwu, and B. Rau. “Region-Based Compilation: An Introduction and Motivation,” *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995.

[17] U. Holzle, L. Bak, S. Grarup, R. Griesemer, and S. Mitrovic, “Java On Steroids: Sun’s High-Performance Java Implementation,” Presentation slides.

[18] U. Holzle, D. Ungar, “Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback,” *Proceedings of the SIGPLAN ‘94 Conference of Programming Language Design and Implementation*, June 1994.

[19] R. Morgan, *Building an Optimizing Compiler*, Digital Press, Boston, MA, 1998.

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

[21] M. Poletto, D. Engler, and M. F. Kaashoek, “tcc: A system for Fast, Flexible, and High-level Dynamic Code Generation,” *Proceedings of the SIGPLAN ‘97 Conference of Programming Language Design and Implementation*, 1997.

[22] M. Poletto, D. Engler, and M. F. Kaashoek, “tcc: A Template-Based Compiler for ‘C”, *WCSSS*, 1996.

[23] S. Richardson and M. Ganapathi, “Interprocedural Optimization: Experimental Results,” *Software—Practice and Experience*, Vol. 10, No. 2, Feb. 1989, p. 149-169.

[24] S. Richardson and M. Ganapathi, “Interprocedural Analysis vs. Procedure Integration,” *Information Processing Letters*, Vol. 32, Aug. 1989, pp. 137-142.

[25] M. D. Smith, “Extending SUIF for Machine-dependent Optimizations,” *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 14–25, January 1996. URL: http://www.eecs.harvard.edu/machsuif.

[26] O. Traub, G. Holloway, and M. D. Smith, “Quality and Speed in Linear-Scan Register Allocation,” *Proceedings of the SIGPLAN ‘98 Conference of Programming Language Design and Implementation*, June 1998.

[27] C. Young, *Path-based Compilation*, Ph.D. thesis, Harvard University, Division of Engineering and Applied Sciences, January 1998.

[28] C. Young and M.D. Smith, “Branch Instrumentation in SUIF,” *Proceedings of the First SUIF Compiler Workshop*, Stanford, CA, January 1996.

Glossary

address space—the memory allocated for a running program to use. A new address space is allocated each time a program is executed. The address space includes not only memory objects, but also a code space, containing the machine instructions for the program.

backedge—an edge in a control-flow graph which goes from a node to one of its ancestors. The existence of a backedge is indicative of a loop.

basic block—a maximal length sequence of instructions that can be entered only from the first instruction and exited only from the last instruction.

callee-saved registers—registers that the calling procedure expects the called procedure to leave unchanged. If a procedure contains instructions which may modify these registers, it must save the original values of the modified registers to memory and restore them before returning.

conditional branch—an instruction which sets the program counter based on whether a particular condition is true or false. If the condition is true, the program counter is set to a given address called the **taken target**. If the condition is false, the program counter is incremented as normal, so that the next instruction executed is the one immediately following the conditional branch in code space (the **fallthrough target**).

constant time—a measure of an algorithm's speed; a process is said to take constant time if the time required for its completion is always fixed.

context switch—a transfer of machine control. Every program running on a machine has one or more **threads** of control flow, all of which maintain their own individual machine states. A context switch is performed by the operating system to replace the state of one thread with the state of another. Since any given thread can exist only in a single address space, a context switch is required to change address spaces.

control-flow graph (CFG)—a directed graph representing the control flow of a program. Nodes in the graph are basic blocks, and edges represent possible paths of control between basic blocks. For instance, a basic block ending in a conditional branch would have two edges coming out of it—one to the basic block whose first instruction is the branch's taken target, and one to the basic block whose first instruction is the branch's fallthrough target. A CFG representing the control flow of an entire procedure contains two special nodes which do not represent basic blocks: the **ENTRY** node and the **EXIT** node. They represent the entry point and exit point of the procedure and are a convenient abstraction for many algorithms which use CFG representations.

dynamically linked library—a pre-compiled set of functions intended for use by many different programs. The code for a dynamically linked library is loaded into a program's address space at run time.

indirect jump—an instruction which resets the program counter to be equal to the value of a given register.

inlining—a process whereby the entire body of a called procedure is placed into the body of the calling procedure, replacing the call instruction. This exposes more code to possible optimizations, including the elimination of some procedure call overhead (such as the saving of callee-saved registers).

instruction scheduling—an optimization which rearranges the order in which instructions will be executed with the goal of issuing the greatest number of instructions in the fewest number of cycles. Instruction scheduling algorithms are very machine-specific, since different architectures have different hardware constraints, rules concerning how many instructions may be issued at the same time, and so forth.

linear time—a measure of an algorithm's speed. A process is said to take linear time if the time required for its completion is directly proportional to the length of its input.

ray tracer—a type of program for drawing images.

register liveness—information about which registers are **live** at a given point in the program. A register is said to be live at a particular point if there is a path from this point along which the current value of the register may be used. If all possible paths either do not use this register, or place a new value in the register before using it, this register is **dead**.

register reallocation—reassigning which values will be held in which hardware registers. At some points, a program needs to keep track of more values than it has hardware registers; those values which cannot be placed in registers must be **spilled** (stored in memory).

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Mike Smith. He presented me with a fascinating problem space to explore and was an active participant in my research. His door was always open to offer guidance, technical advice, and encouragement. I learned a lot from him and enjoyed it all the while.

Along with Mike, Omri Traub and Glenn Holloway provided constant input and support. Omri generously allowed me to share his office and helped keep me on the right track with his suggestions and feedback. Glenn's vast technical knowledge, tireless work ethic, and friendly disposition made him an invaluable resource.

I would also like to mention a few additional people who made direct contributions to the work presented in this thesis. Christian Carrillo collected some of the path statistics used in Chapter 2. Sylvan Clarke wrote early versions of the decoder, encoder, and patcher. Norm Rubin from Digital Equipment Corporation provided a great deal of assistance with Alpha-specific issues; the current encoder is a modification of code provided by Norm.

Finally, I would like to thank my family and friends for their support and encouragement, without which I never would have finished. My brother Matthew's gallant attempt to read this thesis provided a much-needed perspective on how to make it more readable. I would also especially like to thank Becky Weiss for keeping me sane and for taking on the hazardous job of being my first proofreader.