Eliminating Scope and Selection Restrictions in Compiler Optimization

Spyridon Triantafyllis

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

September 2006

Abstract

To meet the challenges presented by the performance requirements of modern architectures, compilers have been augmented with a rich set of aggressive optimizing transformations. However, the overall compilation model within which these transformations operate has remained fundamentally unchanged. This model imposes restrictions on these transformations' application, limiting their effectiveness.

First, procedure-based compilation limits code transformations within a single procedure's boundaries, which may not present an ideal optimization scope. Although aggressive inlining and interprocedural optimization can alleviate this problem, code growth and compile time considerations limit their applicability.

Second, by applying a uniform optimization process on all codes, compilers cannot meet the particular optimization needs of each code segment. Although the optimization process is tailored by heuristics that attempt to *a priori* judge the effect of each transformation on final code quality, the unpredictability of modern optimization routines and the complexity of the target architectures severely limit the accuracy of such predictions.

This thesis focuses on removing these restrictions through two novel compilation framework modifications, Procedure Boundary Elimination (PBE) and Optimization-Space Exploration (OSE).

PBE forms compilation units independent of the original procedures. This is achieved by unifying the entire application into a whole-program control-flow graph, allowing the compiler to repartition this graph into free-form regions, making analysis and optimization routines able to operate on these generalized compilation units. Targeted code duplication techniques can then recover the performance benefits of inlining while limiting code growth. Thus PBE offers a superset of the benefits of inlining and interprocedural optimization, while avoiding both excessive code growth and overly long compile times.

OSE, on the other hand, explores many optimization options for each code segment and selects the best one *a posteriori*. OSE trims the space of options explored through limited use of heuristics, further limits the search space during compiler tuning, and exploits feedback to prune the remaining optimization configurations at compile time. The resulting optimization outcomes are compared through a fast and effective static performance estimator. As a result, OSE is the first iterative compilation technique fast enough and general enough for general-purpose compilation.

Acknowledgements

My graduate research work was supported by National Science Foundation grant CCR-0082630; a grant from the DARPA/MARCO Gigascale Silicon Research Center; several donations from the Intel Corporation; and an Intel Graduate Fellowship. Opinions, findings, conclusions, and recommendations expressed throughout his work are not necessarily the views of the National Science Foundation, DARPA/MARCO, or the Intel Corporation.

I would like to thank my advisor, David August, for his advice and support during my tenure at Princeton. In addition to financial support and day-to-day guidance, David provided the initial ideas that eventually developed into the techniques presented in this thesis. Perhaps more importantly, David gave me the opportunity to work within the Liberty Research Group, a team of like-minded and highly motivated fellow students. In addition to active support in all aspects of my research work, both the high-minded and the tedious, this group offered me an invaluable education in working in a team environment. This experience has already proven priceless in my post-graduate school life.

Many thanks go to Manish and Neil Vachharajani, Ram Rangan, David Penry, Bolei Guo, Guilherme Ottoni, and the rest of my former labmates. By sharing my workload, discussing research and implementation ideas, and offering feedback and guidance on all aspects of my work, they have made this dissertation possible. I would especially like to thank Matthew Bridges and Easwaran Raman for their help in all stages of PBE's development and for sharing with me the enormous task of creating a working research compiler from the ground up. Without their tireless efforts, very little of my dissertation work would have ever come to fruition.

I would also like to thank the Intel Corporation, and especially Neil Glew, Michal Cierniak, Robert Cox, Daniel Lavery, and Carole Dulong for funding my thesis work through an Intel Graduate Fellowship during the 2004-05 academic year, for honing my "real-world" work skills during an internship in the summer of 2003, for offering feedback on various parts of my research, and for providing me access to their proprietary compilation software

and support for its use.

On a more personal note, I owe great gratitude to the friends, both old and new, that provided me with their company and their support throughout my graduate career. Tasos Viglas, George Karakostas and the rest of Greek “old guard” took me under their wing early on, when I was struggling to adjust to a new country and a new work environment. Thanasis Laliotis, Dimitris Dentsoras, Giorgos Skretas, Ioannis Avarmopoulos, Bob Min, Hrijoy Bhattacharjee and many others showed me that hard play can go hand in hand with hard work, and that a graduate student’s life need not be devoid of fun and enjoyment.

Contents

Chapter 1

Introduction

The performance of modern computing systems increasingly depends on the effectiveness of aggressively optimizing compilation. At one end, even mass-market processors increasingly incorporate a multitude of advanced performance features, such as multitiered memory hierarchies, explicitly parallel ISAs, and even multiple on-chip cores. Such features cannot be fully exploited unless an optimizing compiler aggressively targets them. At the other end, object-oriented, strictly typed languages such as C# and Java are fast replacing lower-level programming facilities, while quasifunctional and application-specific languages are gaining in popularity.

Both these trends increase the demands placed on optimizing compilation. In addition to the traditional tasks of simplifying computations and eliminating redundancies, a modern aggressively optimizing compiler must work around inconvenient language features, such as programs' fragmentation into small procedures, while effectively exploiting complex computational resources, exposing instruction-level parallelism (ILP) and/or thread-level parallelism (TLP), and avoiding performance pitfalls such as memory stalls and branch misprediction penalties. As a result, over the last decade, compilers have been enhanced with a multitude of new optimization routines. However, the overall compilation framework in which these routines are applied has changed relatively little. This discrepancy decreases optimization effectiveness, thus hampering a compiler's ability to influence a system's performance. The central focus of this dissertation lies in identifying the ways in which the traditional compilation framework limits optimization efficiency and in proposing ways to remove these limitations.

1.1 Optimization Scope Limitations

Traditionally, most of the analysis and optimization routines incorporated in an optimizing compiler operate within the scope of individual procedures. This scope limitation greatly hampers the ability of these routines to produce efficient code. The original breakup of a program into procedures serves software engineering rather than optimization goals, often making procedures less than ideal optimization units. This problem is only exacerbated by modern software engineering techniques, such as object-oriented programming, which typically encourage small procedures (methods) and frequent procedure calls.

To alleviate the inconvenient optimization scope of individual procedures, most modern compilers employ interprocedural optimization and/or aggressive inlining. Interprocedural analysis and optimization [9, 20, 30, 37, 44, 45, 46, 47] vastly increase the amount and accuracy of information available to the optimizer, exposing many previously unavailable optimization opportunities. However, the compile-time cost of these methods, both in terms of memory usage and time, increases dramatically with program size. As a result, interprocedural optimization is either sparingly applied or omitted entirely from commercial compilers.

Inlining, originally proposed to limit call overhead, copies the bodies of selected procedures into their call sites [3, 6, 10, 18, 24, 49, 53, 54]. This not only exposes more code to analysis and optimization routines, it also allows the optimizer to specialize the callee's code for each particular call site. Unfortunately, the benefits of aggressive inlining come at the cost of extensive code growth. Since the adverse effects of code growth quickly become prohibitive, many limitations are placed on the inlining routine. In particular, inlining is usually limited to frequently executed call sites with small callees.

The optimization scope problem, along with the benefits and shortcomings of existing approaches to it, is discussed in Chapter 2.

1.2 Optimization Decision Limitations

Unfortunately, the limited scope of procedure-based compilation is not the only way in which a traditional compilation framework limits optimization effectiveness. A second limitation stems from the the compiler's difficulty in reaching the correct optimization decisions. Due to the complexity of modern processors, judging the impact of a code transformation on final code quality cannot be achieved by simple metrics such as instruction count or code size. Instead, an optimizing compiler has to carefully balance a set of performance factors, such as dependence height, register pressure, and resource utilization, as well as anticipate dynamic effects, such as cache misses and branch mispredictions.

This task is complicated by the fact that most modern optimization routines usually constitute tradeoffs, improving some performance factors while worsening others. As a result, a successful optimizing compiler must correctly determine where and when to apply each code transformation. In today's compilers, this is usually achieved through the use of *predictive heuristics*, which try to *a priori* judge the impact of an optimization routine on a code segment's final performance. However, a heuristic's task is complicated not only by the complexity of the target platform, but also by the complexity of interactions between optimization routines. As a result, these heuristics lead to wrong optimization decisions in many cases.

To address these limitations, *iterative compilation* [1, 13, 16, 29, 32, 40, 56, 57] has been proposed. Instead of relying on *a priori* predictions, an iterative compiler applies many different optimization configurations on each code segment and decides which one is best *a posteriori*. Previous research indicates that iterative compilation can provide significant performance benefits. However, current iterative compilation techniques incur prohibitive compile-time costs, essentially placing iterative compilation's benefits out of the reach of general-purpose compilers. Chapter 7 deals with the optimization decision problem and with current iterative compilation approaches.

1.3 Contributions

In order to overcome the above restrictions, this dissertation introduces two novel modifications to the traditional compilation framework: Procedure Boundary Elimination (PBE) and Optimization-Space Exploration (OSE).

PBE expands the scope of optimization by unifying an entire application's code into an optimizable program-wide representation, called the *whole-program control-flow graph* (WCFG). By employing an expanded version of *region formation* [22], the compiler can then divide the WCFG into more manageable compilation units. These new compilation units are chosen according to the needs of optimization, regardless of the original procedure boundaries. Since code duplication, such as that caused by inlining, can often increase optimization opportunities, PBE also employs a *targeted code specialization* phase, which causes code growth only where it is likely to produce actual benefits. Of course, the central challenge that this scheme faces is that subsequent optimization and analysis routines must be able to operate on arbitrary portions of the WCFG. This problem was solved by extending and generalizing interprocedural analysis algorithms [47], and by properly encapsulating regions. As a result, PBE offers a superset of the benefits of inlining and interprocedural optimization, while avoiding both excessive code growth and overly long compile times.

PBE is presented in Chapters 3 to 6 of this dissertation. Chapter 3 focuses on the overall PBE compilation framework. Chapter 4 presents the dataflow analysis algorithm that makes PBE compilation possible. This algorithm is based on existing interprocedural analysis approaches, but includes significant modifications that allow it to deal with the nontraditional flowgraphs that PBE produces. Chapter 5 describes how PBE keeps compile time in check through region-based compilation. Finally, Chapter 6 offers an experimental evaluation of PBE. PBE has been previously published in the proceedings of the 2006 Conference on Programming Language Design and Implementation [50].

While PBE deals with the problems of limited optimization scope, OSE is a novel iterative compilation technique that deals with improving optimization decisions. Unlike previous iterative compilation attempts, which suffer from unrealistic compile times and focus on limited portions of the optimization process, OSE is the first iterative compilation technique fast enough and general enough for widespread use in general-purpose compilation. To achieve this, OSE narrows down the space of optimization options explored through judicious use of heuristics. A reasonably sized subarea of this reduced search space is then selected for exploration during a compiler tuning phase. At compile time, OSE exploits feedback from optimization sequences already tried in order to further prune the search on a per-code-segment basis. Finally, rather than measuring actual runtimes, OSE compares optimization outcomes using a fast but effective static performance estimator. The OSE framework is presented in Chapter 8. Chapter 9 provides details of a proof-of-concept implementation of OSE using a commercial compiler and provides detailed experimental results. OSE has been previously published in the proceedings of the 2003 International Symposium on Code Generation and Optimization [52] and in the Journal of Instruction-Level Parallelism [51].

Finally, Chapter 10 summarizes the contributions of this thesis, draws conclusions, and discusses avenues for future research.

Chapter 2

The Optimization Scope Problem

Forcing optimization and analysis routines to operate within individual procedures obviously limits their ability to improve performance. Since procedures are chosen based on criteria unrelated to optimization, their boundaries may conceal significant optimization opportunities. For example, procedure calls within loops can conceal cyclic code from the compiler, hindering both traditional loop optimizations and loop parallelization transformations. Additionally, breaking up a computational task into many small procedures may prevent a scheduling routine from constructing traces long enough to provide sufficient ILP opportunities.

In order to recover optimization opportunities concealed by procedure boundaries, most modern optimizing compilers apply aggressive inlining and, to a far more limited extent, interprocedural optimization. Section 2.1 discusses interprocedural optimization and analysis. Section 2.2 presents existing work on inlining and surveys its benefits and shortcomings. Section 2.3 presents region-based compilation, often used to limit the excessive compile time resulting from over-aggressive inlining. Finally, Section 2.4 presents variants of inlining that can selectively duplicate parts of a procedure.

2.1 Interprocedural Analysis and Optimization

When operating within a single procedure, a dataflow analysis routine generally computes a “meet-over-all-paths” solution to a set of dataflow equations. In contrast to this, a *precise* (or *context-sensitive*) interprocedural analysis routine has to compute a “meet-over-all-*realizable*-paths” solution. A path on an interprocedural control-flow graph is considered realizable if its call and return transitions are in accordance with procedure call semantics. More precisely, on a realizable path a return edge cannot occur without a matching call edge, and call and return edges have to be properly nested.

Sharir and Pnueli [47] are credited with the first systematic approach to interprocedural dataflow analysis. They present the *functional* approach to dataflow analysis, which, with several variations, forms the base of almost all modern interprocedural analysis methods. Because the PBE analysis algorithm is also a derivative of the functional approach, Sharir and Pnueli’s algorithm will be outlined in Chapter 4. A second approach proposed in the same article, called the *call-strings* approach, is easier to understand but leads to much less efficient analyses for most common problems. It is generally used only in approximative interprocedural analysis systems. Sharir and Pnueli [47] also prove that both these approaches converge to a maximal fixed point (MFP) solution for all monotone dataflow analysis problems on bounded semilattices. In the case where the dataflow analysis problem is also distributive, Knoop and Steffen [30] have proven that the MFP solution is also accurate, in the sense that it coincides with the meet-over-all-realizable-paths solution.

The interprocedural dataflow analysis problem is exponential in the general case. However, for the more limited class of analyses typically used in a general-purpose optimizer, including the one presented in this paper, efficient algorithms are available. For locally separable (“bit-vector”) problems, the asymptotic complexity of interprocedural analysis is no greater than that of intraprocedural analysis [31]. By reducing the interprocedural dataflow analysis problem to graph reachability on a properly expanded version of the control-flow graph, Reps et al. [44] were able to provide an efficient interprocedural analysis algorithm for all finite distributive subset dataflow problems.

A discussion of interprocedural analysis would be incomplete without mentioning memory analysis. Indeed, all nontrivial points-to or alias analyses are interprocedural; see [11, 12, 21] among many others. Also, Myers [39] studies classical interprocedural dataflow analysis in conjunction with alias analysis, showing that even bit-vector interprocedural dataflow analysis problems become NP-hard if the source language supports reference parameters. However, a discussion of memory analysis lies outside the scope of this thesis. This is because memory analysis and PBE are orthogonal to each other. Memory analysis is equally necessary and useful in a traditional optimizing compiler as in a PBE compiler. Also, PBE neither helps nor hinders memory analysis.

Using the results of interprocedural analysis, some classical optimization routines can be performed interprocedurally. Morel and Renvoise [37] propose an interprocedural version of partial redundancy elimination. Interprocedural constant propagation, which is relatively fast and effective, has been extensively studied; see for example [9, 20, 45]. Finally, some interprocedural optimizations, such as the link-time register allocation algorithm proposed by Santhanam and Odnert [46], work by propagating problem-specific summary information along the call graph, without having to apply interprocedural dataflow analysis algorithms.

Obviously, interprocedural optimization routines are able to exploit many more optimization opportunities than their intraprocedural counterparts. However, compilers normally apply interprocedural optimization to a very limited extent, and often not at all. This is because of the superlinear complexity of interprocedural analysis and optimization routines, which makes it difficult to apply them repeatedly to entire real-world programs without prohibitive increases in compile time and memory utilization. An additional problem ensues for interprocedural optimizations that may extend variables' live ranges, since data exchange between procedures is normally possible only through parameter-passing or global variables. Adding new parameters or transforming local variables to global both carry costs, which limit the value of such transformations. For this reason, optimizations such as partial redundancy elimination or loop-invariant code motion are rarely applied interprocedurally. PBE overcomes this problem by eliminating special assumptions about procedure boundaries and addresses the problem of excessive compile-time costs through region-based compilation (Chapter 3).

2.2 Inlining

Inline procedure expansion, or simply inlining, eliminates procedure calls by replacing selected call sites with copies of their callees. Originally used to eliminate call overhead, inlining is aggressively applied by many modern compilers in order to expose additional optimization opportunities. Inlining, its performance impact, and the heuristics that drive it have been extensively studied. For example, the high-performance compiler implemented by Allen and Johnson [3] conservatively applied inlining on small static C functions in order to increase vectorization opportunities. A much more aggressive inlining methodology, studied by Hwu and Chang [10, 24], aimed to remove around 60% of a program's dynamic call sites by using a variety of heuristic criteria, some of them profile-driven. Ayers et al. [6] studied a compiler that combined profile-driven inlining with the closely related technique of procedure cloning for greater effect.

The benefits of inlining come from increasing the size of procedures, thus expanding optimization scope, and from allowing code specialization by enabling the compiler to optimize the body of a callee according to a particular call site. This provides significant performance benefits. For example, inlining-induced speedups reported in [10] are 11% on average, reaching up to 46% for some benchmarks. Even greater speedups are reported on SPECint95 benchmarks in [6], with an average of 32% and a maximum of 80%.

Unfortunately, inlining's benefits often come at the cost of excessive code growth. Since traditional inlining can only copy entire procedure bodies, it must duplicate both "hot" and “cold” code, despite the fact that the latter is unlikely to offer any performance benefits. For example, the inlining heuristics in [10] can cause code growth of up to 32%. One undesirable side effect of excessive code growth is that optimization and analysis time may increase significantly, especially if individual procedure bodies grow too large. In [6], for example, inlining causes up to a 100% increase in compile time. In a rather more extreme experiment, Hank et al. [22] report a more than eightfold compile-time increase when 20% of a benchmark’s call sites are inlined. To avoid these pitfalls, commercial compilers usually apply inlining only to frequent call sites with small callees, thus limiting the technique’s applicability and value.

2.3 Region-Based Compilation

Region-based compilation was proposed by Hank et al. [22] in order to cope with the excessive compile-time dilation that aggressive inlining occasionally causes. A region is essentially a compiler-selected, multiple-entry multiple-exit portion of a procedure, which is analyzed and optimized in isolation. This is made possible by properly annotating dataflow information, mainly liveness and points-to sets, on a region’s boundaries, and by teaching the rest of the optimization process to restrict its operation within a single region at a time. Experiments in [22] show that region-based compilation can reduce the time spent in expensive compilation phases, such as register allocation, by more than 90% at almost no cost to performance.

As we will see in Chapter 5, PBE is also a region-based compilation technique, although both the region formation algorithm used and the overall way in which regions are incorporated into the compilation process are significantly different.

2.4 Partial Inlining

Partial inlining alleviates traditional inlining's code growth problems by duplicating only a portion of the callee into the call site. This is achieved by abstracting away infrequent parts of the callee into separate procedures, and inlining only the remaining, frequently executed part.

Way et al. [53, 54] propose a partial inlining framework based on region formation (Section 2.3). This framework uses a profile-guided heuristic to form regions across procedure calls. It then breaks up the parts of callees that lie outside these regions (and therefore are less frequently executed) into new procedures, and inlines just the remaining, frequently executed parts. Compared to full inlining, this framework achieves modest code growth reductions (between 1% and 6%) without sacrificing performance. Implementations of partial inlining in just-in-time compilers, such as the one proposed by Suganuma et al. [49], also have the option of simply deferring the compilation of the callee's cold portions until they are first entered, which may never happen in a typical execution. If the code attempts to access the omitted portions, the corresponding inlining decision is canceled and a recompilation is triggered. This results in a substantial code growth savings of up to 30%, at least when recompilation is not triggered. Other work, such as Goubault [18], proposes less general partial inlining techniques as part of specific optimization or analysis routines.

By providing the compiler with more flexibility as to which parts of the code are duplicated, partial inlining can strike a better balance between code growth and performance improvement. However, this flexibility is limited in several ways. If the cold code has to be repackaged as one or more procedures, only single-entry, single-exit code regions can be excluded from duplication. More general cold regions have to be converted to single-entry single-exit form through tail duplication, which introduces code growth with no performance benefits. Perhaps more importantly, transitions from hot to cold code, which were originally intraprocedural, must be converted to calls and returns. Additionally, any data exchange between these two parts of the code has to be implemented through parameter passing or global variables [54]. This makes these transitions much costlier, which in turn makes partial inlining worthwhile only for procedures containing sizable parts of very infrequently executed code. This restriction is even more pronounced in [49], where a transition from hot to cold code forces recompilation. For these reasons, partial inlining is not broadly used in commercial compilers.

Chapter 3

A Framework for Unrestricted Whole-Program Optimization

In order to overcome the limitations of procedure-based compilation while avoiding the shortcomings of interprocedural optimization and inlining presented in Chapter 2, this chapter introduces *Procedure Boundary Elimination* (PBE), a compilation framework that allows unrestricted whole-program optimization. Section 3.1 gives a general overview of PBE. The overall PBE compilation flow is outlined in Section 3.2. Sections 3.3 and 3.4 present two of PBE's compilation phases, procedure unification and targeted code specialization. The presentation of PBE will continue in Chapter 4, which describes the PBE dataflow analysis algorithm, and in Chapter 5, which describes how PBE can be fitted into a region-based compilation framework.

3.1 Overview

PBE removes the restrictions that a program's division into procedures imposes on optimization. Unlike current methods that address this problem, such as inlining and interprocedural analysis, PBE suffers neither from excessive code growth nor from excessive compile time and memory utilization.

PBE begins with *procedure unification*, which unifies an application's code into an analyzable and optimizable *whole-program control-flow graph* (WCFG). Similar to Sharir and Pnueli [47], this is achieved by joining the individual control-flow graphs (CFGs) of the original procedures through appropriately annotated call and return arcs. To make the WCFG freely optimizable, PBE then takes several additional actions. Among other things, calling convention actions are made explicit, local symbol scopes are eliminated in favor of a program-wide scope, and the stack-like behavior of local variables in recursive procedures is exposed.

Aggressive inlining realizes performance benefits not just by expanding optimization scope, but also by specializing procedure bodies to particular call sites. To recover these benefits, PBE includes *targeted code specialization* (TCS) routines in the optimization process. Such routines duplicate code aggressively enough to obtain significant specialization benefits, while limiting code growth to where it is likely to be beneficial for optimization.

Since PBE allows arbitrary code transformations to be applied across procedure boundaries, optimization and analysis phases are presented with compilation units that are very different from procedures. Among other things, call and return arcs can have a many-to-many correspondence, and former procedure boundaries may no longer be recognizable. Effective optimization requires that such compilation units be accurately analyzed. As described in Chapter 4, the PBE analysis framework achieves this by extending the interprocedural analysis algorithms presented by Sharir and Pnueli [47].

Of course, PBE would be unrealistic if it tried to apply every optimization and analysis routine on the entire program. For this reason, PBE aggressively applies *region formation* [22]. This partitions the WCFG into compiler-selected, arbitrarily shaped subgraphs whose nodes are deemed to be "strongly correlated" according to some heuristic. These partitions are then completely encapsulated, so as to be analyzable and optimizable as separate units. For the purposes of optimization, this new partitioning of the program is preferable to its original breakup into procedures because regions, unlike procedures, are selected by the compiler for the explicit purpose of optimization. To fit region formation into the correctness and performance requirements of PBE, both the region formation heuristics and the way regions are encapsulated were significantly modified from related work. The PBE region formation algorithm is discussed in Chapter 5.

The end result is that PBE obtains a superset of the benefits of both interprocedural optimization and inlining, while avoiding excessive compile-time dilation, unnecessary code growth, and scalability limitations. As Chapter 6 shows, PBE is able to achieve better performance than inlining with only half the code growth. By allowing the compiler to choose each compilation unit's contents, PBE ensures that each unit provides a sufficiently broad scope for optimization, reducing the need for program-wide analysis. Additionally, the compiler can control the size of these compilation units (regions) so as to strike a reasonable balance between optimization effectiveness and compile-time dilation. By enabling fine-grained specialization decisions, PBE avoids unnecessary code growth. PBE provides extra freedom to both region formation and optimizations by enabling optimization phases to deal with free-form compilation units, thus realizing new optimization benefits that were not available through either inlining or interprocedural optimization.

3.2 PBE Compilation Flow

The overall flow of the compilation process in PBE can be seen in Figure 3.1. A PBE compiler begins by applying the following three phases:

Unification This phase merges the control-flow graphs (CFGs) of individual procedures into a single, whole-program control-flow graph (WCFG) and removes all assumptions about calling conventions and parameter passing mechanisms.

Region Formation This phase breaks up the WCFG into compiler selected optimization units, or *regions*, and encapsulates regions appropriately so that they can be analyzed and optimized independently.

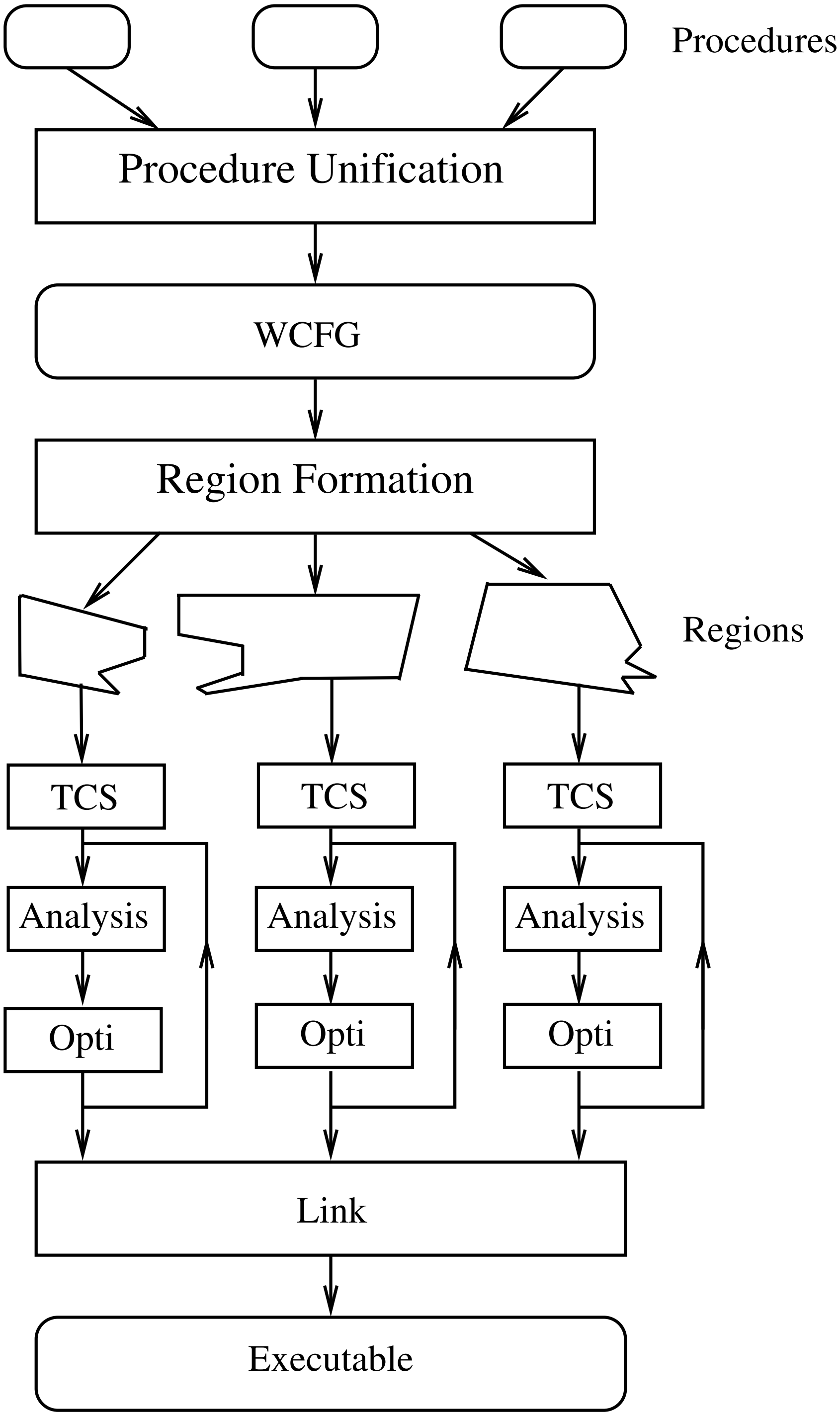
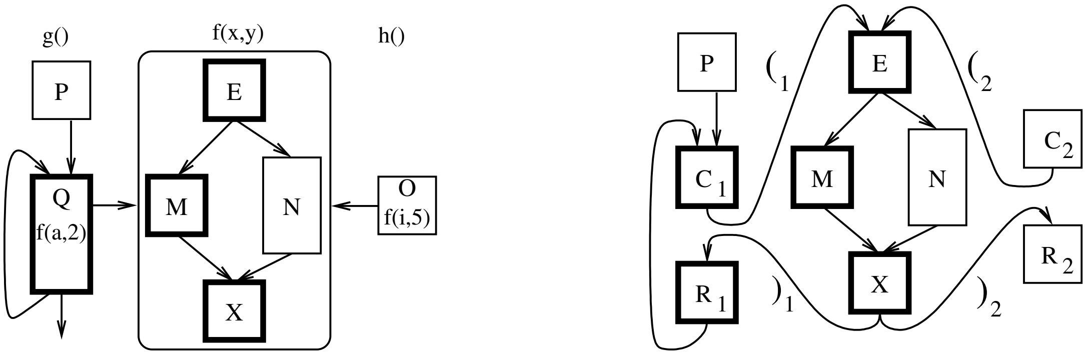


Figure 3.1: Overview of PBE compilation flow. Rectangles represent compilation phases, whereas rounded and irregular boxes represent code units.



(a) Before unification

(b) After unification

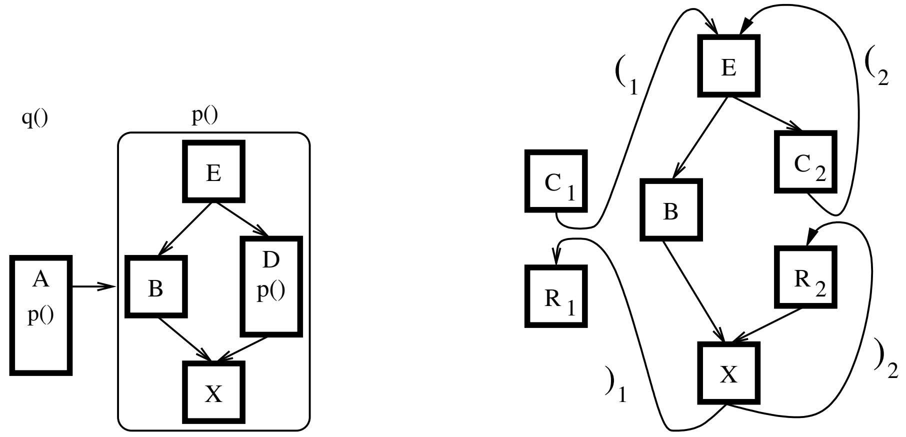
Figure 3.2: A code example (a) before and (b) after unification.

Targeted Code Specialization (TCS) This phase is applied separately within each region, as part of the overall optimization process. It identifies sites in the region where code specialization is likely to provide optimization opportunities and duplicates code accordingly. Like any other optimization routine, TCS methods could be applied on the entire WCFG, before region formation. However, this would be undesirable for compile time and scalability reasons, as explained in Chapter 5.

The above three phases produce compilation units that bear little resemblance to procedures. Therefore, the most important component of a PBE compiler is an optimization and analysis process that can handle these constructs. The analysis algorithm presented in Chapter 4 forms the centerpiece of this process.

3.3 Procedure Unification

The purpose of procedure unification is to combine the individual control-flow graphs of a program's procedures into a whole-program control-flow graph (WCFG). This requires joining the CFGs of individual procedures with control-flow arcs that represent call and return transitions. Due to the semantics of procedure invocation, call and return arcs carry special semantic constraints. On any path corresponding to a real execution of the pro-



(a) Before unification

(b) After unification

Figure 3.3: Recursive procedure (a) before and (b) after unification.

gram, successive calls and returns must appear in a stack-like fashion, with a return always matching the call that is topmost on the stack at each point. A call arc and a return arc are said to *match* if they come from the same original call site. Following the conventions of the interprocedural analysis bibliography [44], these semantic constraints are represented by annotating call and return arcs with numbered open and close parentheses respectively. This notation is convenient, as the matching between calls and returns on a valid program path exactly follows the standard rules for parenthesis matching.

More specifically, unification begins by assigning unique numbers to all of the program's call sites. Let C_i be a call site for a procedure p, and let R_i be the corresponding return site. Let E_p be the entry node (procedure header) of p, and X_p be the exit node (return statement) of p. In the WCFG, this call is represented by two interprocedural arcs: a call arc $C_i \xrightarrow{(_i} E_p$ and a return arc $X_p \xrightarrow{)_i} R_i$.

These concepts are illustrated by the example in Figure 3.2. Figure 3.2a shows a small procedure f with two call sites, one in a "hot" loop in procedure g, and a less frequently executed one in procedure h. In this figure, and in the examples that follow, rectangular boxes represent basic blocks. Frequently executed basic blocks are shown with bold lines. Figure 3.2b shows the same code after unification has been applied. As the figure illustrates, the valid program path $C_1 \xrightarrow{(_1} E \rightarrow M \rightarrow X \xrightarrow{)_1} R_1$ contains the matching parentheses $(_1\ )_1$, whereas the invalid path $C_2 \xrightarrow{(_2} E \rightarrow N \rightarrow X \xrightarrow{)_1} R_1$ contains the mismatched parentheses $(_1\ )_2$. The use of parenthesis annotations in analysis will be presented in Chapter 4.

Perhaps more interesting is the example in Figure 3.3, which shows the result of applying procedure unification to a recursive procedure. After unification (Figure 3.3b), a single self-recursive call appears as two loops: one loop for the recursive call, whose back edge is $C_2 \xrightarrow{(_2} R_2$, and one loop for the recursive return, whose back edge is $R_2 \rightarrow X$. Moreover, it is easy to see that both these loops are natural, since their headers dominate their back edges. In later compilation phases, both these loops can benefit from optimizations traditionally applied to intraprocedural loops, such as loop invariant code motion, loop unrolling, and software pipelining. Although inlining can achieve effects similar to loop unrolling by inlining a recursive procedure into itself, and certain interprocedural optimization methods can achieve results similar to loop invariant code motion, the way recursion is handled in PBE is clearly more general.

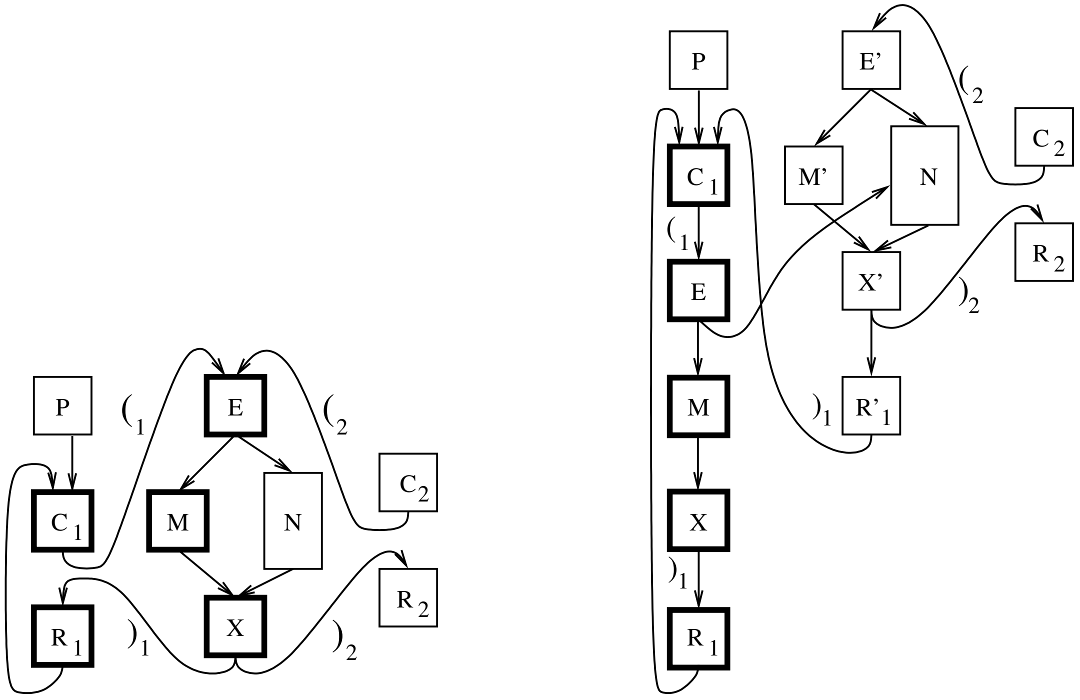
Apart from their matching and nesting constraints, call and return arcs also have other implied semantics very different from those of intraprocedural arcs. Traversing a call arc normally implies saving the return address, setting up a new activation record for the callee, moving actual parameter values into formal parameters, and generally taking any other action dictated by the calling convention. Interprocedural optimizations respect these semantics and work around them appropriately, although this complicates or even hinders their application. In that case, respecting the calling convention is necessary, since these routines must preserve a program's division into procedures for later compilation phases. PBE takes the opposite approach. Since the rest of the compilation process does not depend on procedures and the conventions accompanying them, all these implicit actions are made explicit in the program's intermediate representation (IR). This frees further optimization routines from the need to navigate around calling conventions. For example, a redundancy elimination routine can now freely stretch the live range of a variable across a call arc, without having to convert that variable into a parameter. As an added benefit, the compiler can now optimize those actions previously implicit in calls with the rest of the code, thus reducing their performance impact.

In order to make call and return arcs behave more like normal arcs, unification applies the following transformations on the program's intermediate representation (IR):

- A single, program-wide naming scope is established for variables and virtual registers. This is accomplished by renaming local variables and virtual registers as necessary. To avoid violating the semantics of recursive calls, placeholder save and restore operations are inserted before each recursive call and after each recursive return. (Recursive calls and returns are simply those that lie on cycles in the call graph). These operations are annotated with enough information to allow the code generator to expand them into actual loads and stores to and from the program stack.

- Sufficient fresh variables are created to hold the formal parameters of every procedure. Then the parameter passing is made explicit, by inserting assignments of actual parameter values to formal parameter variables at every call site. The return value is handled similarly. Later optimizations, such as copy and constant propagation and dead code elimination, usually remove most of these assignments.

- Call operations are broken up into an explicit saving of the address of the return node and a jump to the start of the callee procedure. This is done because a call operation always returns to the operation immediately below it. This in turn makes it necessary for a return node to always be placed below its corresponding call node. By explicitly specifying a return address, call and return nodes can move independently of each other. This ensures that optimizations such as code layout and trace selection can operate without constraints across call and return arcs. It also allows code specialization routines to duplicate call sites without having to duplicate the corresponding



(a) Before superblock formation

(b) After superblock formation

Figure 3.4: The code of Figure 3.2b (a) before and (b) after superblock formation.

return sites and vice versa.

- Any actions pertaining to the program stack, such as allocating activation frames, are made explicit in a similar way.

After unification concludes, further code transformations are free to operate on the whole program, without regard to the program's original procedure boundaries (except for the distinction between realizable and unrealizable paths). Eventually, the optimization process will result in code that looks very different from traditional, procedure-based code.

3.4 Targeted Code Specialization

In order to match the performance benefits of aggressively inlining compilers, a PBE compiler must do more than choose the right scope for optimization, as inlining's benefits come not only from increased optimization scope, but also from code specialization. Unlike inlining, PBE does not cause any code growth while forming compilation units. The code growth budget thus freed can now be devoted to more targeted code duplication techniques, which can recover the specialization benefits of inlining with much more modest code growth.

In general, a code specialization technique duplicates selected code segments in order to break up merge points in the CFG. These control-flow merges generally restrict optimization by imposing additional dataflow constraints. After duplication, on the other hand, each one of the copies can be optimized according to its new, less restrictive surroundings. In this sense, both full and partial inlining are code specialization techniques. Several intraprocedural code specialization methods have also been proposed [7, 14, 23, 25, 36], usually in the context of scheduling. Some of these methods can be adapted to the specialization needs of PBE. Indeed, PBE gives such methods new freedom, since it allows them to work across procedure boundaries.

3.4.1 Superblock Formation

Superblock formation [25] is perhaps the simplest and most powerful specialization method. Using profile information, superblock formation selects “hot” traces and eliminates their side entrances through tail duplication. This tends to organize frequently executed areas of the code into long, straight-line pieces of code, which are particularly well suited to both classical optimization and scheduling. In the context of PBE, superblock formation can freely select traces containing call and return arcs, which significantly increases its impact. The effect of applying superblock formation to the code example in Figure 3.2b (repeated for convenience in Figure 3.4a) can be seen in Figure 3.4b. Excessive code growth during superblock formation can be avoided by setting a minimum execution threshold w of blocks to use in superblock formation, a limit a to the relative profile weight of branches followed, and an overall code growth limit b. In the experimental compiler presented in

Chapter 6, w was set to 100 and a was set to 80%. The code growth factor b was set to 50%, although superblock formation usually stays well below this limit. This occurs for several reasons. First, the execution threshold prevents superblock formation on cold code. Second, unbiased branches fall below a, limiting the scope of superblocks. Finally, because superblocks are acyclic, backedges form a natural end point.

3.4.2 Area Specialization

Sometimes it makes sense to duplicate portions of the code that are more complicated than a trace. For example, we may want to specialize an entire loop, or both sides of a frequently executed hammock. For this reason, the PBE compiler presented in Chapter 6 also uses a method called *area specialization*. Like superblock formation, this method is purely profile-driven.

Area specialization begins by identifying an important CFG arc leading to a merge point. It then selects a subgraph of the CFG beginning at the merge point. That subgraph is duplicated, so that the chosen link has its own copy of the subgraph.

Let A be the *duplication area*, i.e. the set of basic blocks selected for duplication. The *frontier* F of the duplication area comprises all basic blocks that do not belong to A and have an immediate predecessor in A. That is:

$$F = \{b \mid b \notin A \wedge \exists a : a \in A \wedge a \rightarrow b \text{ is a CFG edge}\}$$

Each block b in F is assigned a frontier weight FW, which is the sum of the profile weights of control-flow arcs beginning inside the duplication area and ending at b. That is:

$$FW(b) = \sum_{a \in A} W(a \rightarrow b)$$

where $W(x)$ is the profile weight of x.

The algorithm that selects the duplication area proceeds as follows: First, the area contains only the merge point m. In a series of repeated steps, the area is expanded by adding the frontier block b for which $FW(b)$ is maximum. The expansion stops when $FW(b)/W(m) < \alpha$, where α is a tuning parameter. In the experimental evaluation of Chapter 6, $\alpha = 0.1$. To avoid excessive code growth, the entire area specialization phase stops when it duplicates more than a certain percentage β of the region's code. In Chapter 6, $\beta = 50\%$.

The only remaining issue is to choose the merge point arcs for which duplication areas are selected. Although this selection can be done in many ways, for the evaluation of Chapter 6 we chose to consider only call arcs. In a sense, this makes area specialization work like a generalized version of partial inlining. We chose this approach mainly because it makes the comparison between PBE and inlining more straightforward.

3.4.3 Other Specialization Methods

Although the experimental PBE compiler presented in this paper only employs area specialization and superblock formation, any other specialization method that has been proposed in the intraprocedural domain can also be applied within the PBE framework. Bodik et al.'s work on complete redundancy elimination [7] and Havanki's generalization of superblocks to tree-like form [23] seem especially promising in this respect. By being able to work across procedure boundaries, any such specialization method is likely to increase its impact in PBE.

Chapter 4

PBE Analysis

A *context-insensitive* dataflow analysis routine could analyze the WCFG without taking the special semantics of calls and returns into account. However, the analysis results thus produced would be too conservative. For example, such an analysis routine would conclude that any definitions made in block P of Figure 3.4b could reach block R_2, although no valid program path from P to R_2 exists. In preliminary trials, we found that the inaccuracies caused by such an analysis approach have a serious detrimental effect on several compilation phases, especially register allocation. In one such experiment, a unified version of the benchmark `grep` would grow from 837 to 3400 instructions when processed by the IMPACT compiler's traditional register allocator, due to the excessive number of register spill and fill operations.

For this reason, PBE uses a *context-sensitive* analysis algorithm, derived from the *functional approach* to interprocedural analysis [47]. To put the PBE analysis algorithm in context, Section 4.2 walks through the steps of this approach. Both Section 4.2 and the preceding Section 4.1 closely follow the presentation in [47], with the exception that some of the terminology and conventions have been modernized according to [44]. Based on this, Section 4.3 describes the PBE analysis algorithm, which deals with the new challenges presented to the compiler by PBE flowgraphs while staying as close as possible to the traditional interprocedural analysis algorithm presented in Section 4.2, Finally, Section 4.4 summarizes this chapter's contributions.

4.1 Terminology and Background

To simplify the discussion, the rest of this chapter will concentrate on *forward* dataflow problems, such as dominators or reaching definitions. Backward dataflow problems, such as liveness or upwards-exposed uses, can be treated in an entirely symmetric way.

A *dataflow framework* is defined by a pair $(\mathcal{L}, \mathcal{F})$, where $\mathcal{L}$ is a semilattice of *dataflow values* and $\mathcal{F}$ is a space of $\mathcal{L} \to \mathcal{L}$ functions. Let $\sqcup$ denote the semilattice operation of $\mathcal{L}$, heretofore called a "meet". We assume that $\mathcal{L}$ contains a smallest element, $\perp$, normally corresponding to the "empty" dataflow value, and a largest element, $\top$, normally corresponding to the "undefined" dataflow value. Further, $\mathcal{L}$ is assumed to be closed under functional composition and meet, to contain an identity map $\mathbf{id}_{\mathcal{L}}$, and to be monotone, i.e. such that for each function $f \in \mathcal{F}$, $x \leq y$ implies $f(x) \leq f(y)$. For the dataflow analyses that are of interest to this thesis, $\mathcal{L}$ will also be finite. Further, $\mathcal{F}$ will distribute over $\sqcup$, meaning that for each $f \in \mathcal{F}$, $f(x \sqcup y) = f(x) \sqcup f(y)$.

Given a CFG G, we associate with each node $n \in G$ a *transfer function* $f_n \in \mathcal{F}$, which represents the change in dataflow as control passes through n. The notion of transfer functions can be trivially extended from single nodes to paths: if $p = (n_1, ..., n_k)$ is a path in G, then the transfer function of p is $f_p = f_{n_k} \circ \cdots \circ f_{n_1}$. With the transfer functions given, the goal of dataflow analysis is to calculate the dataflow values at the entry and exit of each node, denoted as in_n and out_n respectively. To facilitate the following discussion, we will use $\text{path}_G(m, n)$ to denote the (possibly empty) set of all paths in G between nodes m and n.

Given a program comprising a set of procedures $P_1, P_2, \ldots, P_k$ with corresponding CFGs $G_{P_1}, G_{P_2}, \ldots, G_{P_k}$, the program's *interprocedural control-flow graph* (ICFG) G^* can

$$\begin{array}{lll} SL & \rightarrow \epsilon & \\ SL & \rightarrow (_i \; SL \;)_i & \text{for all } i \\ SL & \rightarrow SL \;\; SL & \end{array}$$

(a)

$$\begin{array}{lll} RP & \rightarrow R \quad SL \quad C & \\ RP & \rightarrow C \quad SL & \\ RP & \rightarrow SL \quad R & \\ R & \rightarrow \epsilon & \\ R & \rightarrow)_i \; R & \text{for all } i \\ C & \rightarrow \epsilon & \\ C & \rightarrow C \; (_i & \text{for all } i \end{array}$$

(b)

Figure 4.1: Grammars for (a) same-level realizable paths and (b) realizable paths.

be constructed by combining the procedures' CFGs in much the same way that PBE's procedure unification begins: by linking each call site c with its callee's entry node e through a *call edge* $c \xrightarrow{(_i} e$, and the callee's exit x with the corresponding return site r through a *return edge* $x \xrightarrow{)_i} r$, where i is an index unique to that call site – callee combination. Additionally, it is convenient to add a *summary edge* $c \xrightarrow{S_i} r$ between the call site and the corresponding return site. (For indirect call sites, which may have more than one callee, we draw one summary edge per callee). The entry e_{main} of the program's main procedure is the entry node of G^*.

A path between two nodes $m, n \in G^*$ is called a *same-level realizable path* iff the string of call and return annotations along its edges can be generated by the grammar shown in Figure 4.1a. Intuitively, a same-level realizable path is a path that begins in a procedure and ends in the same procedure, possibly traversing callee procedures along the way. The existence of a same-level realizable path between nodes m and n will be written as: $m \overset{SL}{\rightsquigarrow} n$. It is easy to see that $m \overset{SL}{\rightsquigarrow} n$ iff there is a path consisting of normal edges and summary edges, but no call or return edges, between m and n.

Given the above definition of same-level realizable paths, we can define more general *realizable paths* as follows: A path between m and n is realizable iff the string of call and return annotations along its edges can be generated by the grammar shown in Figure 4.1b. Essentially, a realizable path may contain unmatched calls and returns, but no *mis*matched calls and returns. It is easy to see that the above definition of realizable paths exactly captures the subset of paths in G^* that do not violate procedure call semantics. All other paths in G^* are unrealizable; they cannot occur in any valid program execution. We will use $\text{rpath}_{G^*}(m, n)$ to denote the set of all realizable paths between nodes $m, n \in G^*$.

4.2 Interprocedural Dataflow Analysis: The Functional Approach

The goal of an intraprocedural dataflow analysis routine on a CFG G with entry node e is to calculate a *meet-over-all-paths* solution, in which the dataflow value at a node n summarizes the dataflow along all paths from e to n:

$$in_n = \bigsqcup_{p \in \text{path}(e,n)} f_p(\bot)$$

Kam and Ullman [27] have shown that, for distributive dataflow problems, this solution can be computed by iteration over the following "local" equations:

$$\begin{aligned}
in_e &= \bot \\
in_n &= \bigsqcup_{(m,n) \in G} out_m && \text{for all } n \in G, n \neq e \\
out_n &= f_p(in_n) && \text{for all nodes}
\end{aligned}$$

In contrast, a context-sensitive interprocedural dataflow analysis algorithm has to calculate a *meet-over-all-realizable-paths* solution, that is:

$$in_n = \bigsqcup_{p \in \text{rpath}_{G^*}(e_{\text{main}},n)} f_p(\bot)$$

Since realizability is a property of the entire path, rather than a property of individual nodes, this solution cannot be arrived at using local calculations only. The obvious way to calculate this is to annotate each piece of dataflow information on a CFG node with the path through which it has propagated. This is roughly the solution provided by the *call-strings* approach to interprocedural analysis, which was also presented in [47]. However, this approach is not generally used in precise interprocedural dataflow analyses because it leads to algorithms with exponential complexity even for simple dataflow problems.

A more promising approach, termed the *functional* approach to interprocedural analysis, works by calculating the transfer functions of entire procedures. After this calculation is done, the dataflow solution can be determined intraprocedurally: the dataflow value of a return site can be calculated by applying the callee's transfer function to the dataflow value of the corresponding call site. More precisely, the algorithm operates in the following two steps:

Step 1: Procedure transfer functions

Let P be a procedure with CFG G_P, entry node e_P and exit node x_P. For convenience, let us assume that G_P contains all relevant summary edges, as defined in Section 4.1 (but, obviously, no call or return edges).

For two nodes $m, n \in G_P$, let $\phi_{(m,n)}$ be the transfer function of all paths between m and n, that is:

$$\phi_{(m,n)} = \bigsqcup_{p \in \text{path}_{G_P}(m,n)} f_p$$

Obviously, the transfer function of the entire procedure will be $\phi_P = f_{x_P} \circ \phi_{(e_P,x_P)}$. We can calculate the transfer functions $\phi_{(e_p,n)}$ for every node $n \in G_P$ by iterating over the formulae:

$$\phi_{(e_P,e_P)} = \mathbf{id}_{\mathcal{L}}$$

$$\phi_{(e_P,n)} = \bigsqcup_{(m,n) \in G_P} h_{(m,n)} \circ \phi_{(e_P,m)}$$

where:

$$h_{(m,n)} = \begin{cases} \phi_Q & \text{if } (m,n) \text{ is a summary edge with callee } Q \\ \mathbf{id}_{\mathcal{L}} & \text{if } (m,n) \text{ is a normal edge} \end{cases}$$

This iteration has to be done on all procedures simultaneously, since the transfer function of each procedure depends on the transfer functions of its callees.

Step 2: Node dataflow values

Using the transfer functions calculated in the above step, the dataflow values of nodes can be calculated by iteratively applying the following formulae:

$$\begin{aligned} in_{e_{\text{main}}} &= \bot \\ in_{e_P} &= \bigsqcup_{c:\ \text{call site of } P} out_c && \text{for all procedures } P \\ in_n &= \phi_{(e_p,n)}(in_{e_P}) && \text{for all } n \in G_P \\ out_n &= f_n(in_n) \end{aligned}$$

Correctness and Efficiency

As mentioned in Section 2.1, the above algorithm has been proven to be correct for a broad class of dataflow analysis problems [30]. The efficiency of the algorithm mainly depends on Step 1 above: intuitively, if the dataflow problem's transfer functions can be represented in a way that makes compositions and meets of such functions easy to compute, then the algorithm is efficient. For the particular case of locally separable dataflow problems, transfer functions can be represented as two bit vectors (the GEN and KILL sets). As a result, computing the meet of two transfer functions is only twice as expensive as computing the meet of two analysis values. For this subclass of problems, it turns out that the functional interprocedural analysis algorithm is asymptotically no more expensive than a context-insensitive or intraprocedural algorithm; on an ICFG with E edges and a dataflow problem with D-length bit-vector values, the algorithm's complexity is $O(ED)$ [31].

4.3 PBE Analysis

PBE analysis faces several challenges that are not present in classical interprocedural analysis and therefore are not handled by the algorithm presented in Section 4.2. Since optimization routines are free to operate along call and return arcs, these arcs may be eliminated, duplicated, or moved. Thus the matching between call and return arcs will generally be a many-to-many relation. For example, in Figure 3.4b, both return arcs $X \xrightarrow{)_1} R_1$ and $X' \xrightarrow{)_1} R_1'$ match the single call arc $C_1 \xrightarrow{(_1} E$. This complicates the semantics of call and return arcs in a fundamental way. For example, the path $E \to N \to X' \to R_2$ in Figure 3.4b is clearly not realizable, despite the fact that it contains no mismatching parenthesis annotations. Moreover, the free movement of instructions across call and return arcs and the free movement of these arcs themselves destroy the notion of procedure membership. Thus dividing the WCFG into procedure-like constructs for the purposes of analysis will itself be a task for the analysis algorithm.

The PBE analysis algorithm presented here meets these challenges, while staying as close as possible to the algorithm described in Section 4.2. The algorithm is presented in two parts. Section 4.3.1 describes an analysis prepass that performs calculations on the control-flow graph only. This forms the input to the main dataflow analysis algorithm, presented in Section 4.3.2.

4.3.1 WCFG Calculations

The first task of the PBE analysis algorithm is to properly classify the WCFG's nodes, to construct procedure-like sets of nodes (called *contexts*), and to draw summary edges. Since the actions presented below are independent of the specific dataflow problem, the results of this phase can be reused for multiple analysis passes. This phase of the analysis only needs to be rerun when the WCFG's nodes are rearranged.

Step 1: CFG node classification

WCFG nodes are classified according to whether they are sources or destinations of call or return arcs. Thus, a node that is the source of at least one call arc is a *call site*. A node that is the destination of at least one return arc is a *return site*. A node that is the destination of at least one call arc is a *context entry*. A node that is the source of at least one return arc is a *context exit*. In addition, the program entry e_{main} will also be considered a context entry. Similarly, the program exit x_{main} will be considered a context exit. Note that these definitions are not mutually exclusive. For example, the special unmatched-call node C used in region encapsulation (Section 5.2) is both a context entry and a call site. Context entries and exits will play similar roles with those of procedure entries and exits in classical interprocedural analysis.

Step 2: Creating context-defining pairs

A pair of nodes (e, x) is called a *context-defining pair* (CDP) if e is a context entry, x is a context exit, and at least one of the call arcs of e matches with some return arc of x. That is, (e, x) is a CDP iff there exists a pair of edges $c \xrightarrow{(_i} e$ and $x \xrightarrow{)_i} r$ for some value of i. In this step, the PBE analysis algorithm identifies and stores such pairs. For the rest of the algorithm, CDPs and the contexts they define (see Step 4) will roughly play the role of procedures. In this spirit, we call the node c above a *call site* of (e, x) and r a *return site* of (e, x) corresponding to call site c. Additionally, the special pair $(e_{\text{main}}, x_{\text{main}})$ will also be considered a CDP.

Step 3: Drawing summary edges

From here on, a path containing normal edges and summary edges, but no call or return edges, will be referred to as a *same-context* path (symbol: $\overset{SC}{\rightsquigarrow}$). The notion of same-context paths is similar to that of same-level paths in Section 4.2. A CDP (e, x) is called *proper* if there is a same-context path $e \overset{SC}{\rightsquigarrow} x$. (Generally, the vast majority of CDP's are proper. Improper CDP's usually arise when some optimization, especially branch constant folding, removes CFG edges, possibly due to constant propagation across former procedure boundaries.) For each proper CDP, we will create *summary edges* leading from call sites of (e, x) to the corresponding return sites. More formally, if (e, x) is a proper CDP, then we will create a summary edge $c \xrightarrow{S_i} r$ for every pair of edges $c \xrightarrow{(_i} e$ and $x \xrightarrow{)_i} r$.

The PBE analysis algorithm discovers proper CDPs and draws summary edges by repeatedly running a reachability algorithm that discovers which nodes are reachable from each context entry along same-context paths. If a context exit x is reachable from a context entry e, we check to see if a CDP (e, x) exists. If it does exist, this CDP is marked as proper and the corresponding summary edges are drawn. The reachability algorithm is rerun after the addition of the new summary edges, possibly leading to more proper CDPs being discovered and new summary edges being drawn. This process has to be repeated until it converges.

Step 4: Discovering context membership

Each CDP (e, x) defines a *context* $CT_{e,x}$. We will say that a node n *belongs* to a context $CT_{e,x}$ iff there are same-context paths $e \overset{SC}{\rightsquigarrow} n$ and $n \overset{SC}{\rightsquigarrow} x$. Obviously, the contexts of improper CDPs will be empty. For the PBE analysis algorithm, context membership roughly corresponds to procedure membership in the classical interprocedural analysis algorithm. (Note however that a node can belong to more than one context). Since forward reachability from context entries has already been calculated in the previous step, a similar backward-reachability pass from context exits is run to determine reachability from context exits.

Context membership can then be easily determined by combining these two reachability results.

Step 5: Building the context graph

Just as call relationships between procedures can be represented in a call graph, reachability relationships between CDPs give rise to a context graph. A directed edge $CT_{e_1,x_1} \rightarrow CT_{e_2,x_2}$ means that there is a call site c and a return site r such that both c and r belong to CT_{e_1,x_1} and there is a call edge $c \xrightarrow{(_i} e_2$ and a matching return edge $x_2 \xrightarrow{)_i} r$. The CDP graph can be easily calculated by going through the call and return edges in the WCFG and combining them with the context membership information from the previous step. The context CT_{main}, corresponding to the CDP $(e_{\text{main}}, x_{\text{main}})$, will be the entry node of the context graph.

4.3.2 Problem-Specific Calculations

Given the results of the prepass in Section 4.3.1, the remainder of the PBE analysis algorithm is similar to the interprocedural analysis algorithm presented in Section 4.2.

Step 1: Context transfer functions

For every context $CT_{e,x}$, this step calculates a transfer function $\phi_{CT_{e,x},n}$ from the entry e to every node $n \in CT_{e,x}$ using only paths in that context. More formally, let $G_{CT_{e,x}}$ be the WCFG subgraph containing all the nodes in $CT_{e,x}$ along with all the normal edges and summary edges that connect them (but no call or return edges). Then:

$$\phi_{CT_{e,x},n} = \bigsqcup_{p \in path_{G_{CT_{e,x}}}} f_p$$

Note that a node n may be contained in multiple contexts with the same entry e; that's why the full context has to be specified in the definition of ϕ above. The transfer function of the entire context $CT_{e,x}$ can be defined as:

$$\phi_{CT_{e,x}} = f_x \circ \phi_{CT_{e,x},x}$$

Similarly to Step 1 of Section 4.2, these transfer functions can be calculated by iteratively applying the formulae:

$$\begin{aligned}\phi_{CT_{e,x},e} &= \mathbf{id}_{\mathcal{L}} \\ \phi_{CT_{e,x},n} &= \bigsqcup_{(m,n)\in G_{CT_{e,x}}} h_{m,n} \circ \phi_{CT_{e,x},m}\end{aligned}$$

where:

$$h_{m,n} = \begin{cases} \phi_{CT_{e',x'}} & \text{if } (m,n) \text{ is a summary edge with “callee” } CT_{e',x'} \\ \mathbf{id}_{\mathcal{L}} & \text{if } (m,n) \text{ is a normal edge} \end{cases}$$

This iteration has to be performed on all contexts simultaneously.

Step 2: Computing node values

In a way similar to the algorithm in Section 4.2, the transfer functions calculated above can be used to calculate the analysis values at all nodes. The only extra complication is that a node can belong to more than one context. The iterative formulae are:

$$\begin{aligned} in_{e_{\text{main}}} &= \bot \\ in_e &= \bigsqcup_{c:\ \text{call site of a } CT_{e,x} \text{ for some } x} out_c && \text{for all context entries } e \\ in_n &= \bigsqcup_{CT_{e,x}:\ n\in CT_{e,x}} \phi_{(CT_{e,x},n)} in_e && \text{for all other nodes} \\ out_n &= f_n(in_n) && \text{for all nodes} \end{aligned}$$

4.4 Conclusion

Although interprocedural analysis algorithms have been presented in the past, these algorithms always assume that the code being analyzed conforms to a “traditional” procedural structure. This in turn limits the compiler’s ability to optimize freely across procedure boundaries. By using the generalized analysis algorithm presented above, PBE avoids these limitations. This algorithm draws heavily on existing work on interprocedural analysis, but contains several key additions, namely the prepass in described in Section 4.3.1 and the modified equations in Section 4.3.2. This algorithm forms the centerpiece of the PBE compiler presented in Chapter 6.

Chapter 5

Region-Based PBE

PBE, as presented in Chapters 3 and 4, could operate on the entire program. Indeed, this would enable code transformations to achieve their maximum performance impact. However, such an approach would not be scalable to even modestly sized programs. This is because most optimization and analysis routines are super-linear, causing compile time and memory utilization to increase very fast with compilation unit size.

Region formation solves this problem by breaking up the WCFG into more manageable *regions*, which are then analyzed and optimized in isolation. Although breaking up the program into regions is bound to cause some performance loss, the compiler is free to decide the size and contents of regions according to its optimization needs. Therefore, it is reasonable to expect that PBE regions will be superior compilation units to the program's original procedures, which are chosen according to criteria unrelated to optimization. Indeed, as seen in Section 2.3, previous research indicates that performance loss due to region formation is minimal.

Section 5.1 describes the region formation heuristic used by our initial implementation of PBE. Section 5.2 describes a region encapsulation method that enables PBE's later phases to handle each region in isolation.

5.1 Region Formation

This section describes the profile-driven region formation heuristic used in the initial implementation of PBE (Chapter 6). This heuristic is very different from the one originally proposed by Hank et al. [22]. Note, however, that the PBE technique does not depend on any particular region formation heuristic. Simpler or more sophisticated heuristics can be used, depending on a compiler's specific needs.

The region formation heuristic presented here has two basic goals. The first is to produce regions whose size is neither too much above nor too much below a user-specified size target S. This is because regions that are too big may overburden the optimizer, while regions that are too small will expose too few optimization opportunities. Second, transitions between regions should be as infrequent as possible. This is because a transition between regions has some runtime overhead, much like the overhead that calls and returns incur in procedure-based programs. This overhead comes both from unrealized optimization opportunities spanning the transition and as a consequence of region-based register allocation.

The first phase of the region formation heuristic is a greedy clustering algorithm. The basic blocks of the WCFG are divided into clusters. The size of a cluster is the total number of instructions contained in its constituent basic blocks. These clusters are connected with undirected weighted edges. The weight assigned to an edge between two clusters is the sum of the profile weights of the real WCFG edges between blocks in the two clusters.

At the start of the formation process, each individual basic block will be in a separate cluster. Clusters are then repeatedly joined by examining the intercluster edge with the highest weight. If the combined size of the two clusters it connects is less than the size target S, then the two clusters are joined, with edges and edge weights being updated accordingly. This phase of the algorithm terminates when no more clusters can be joined without exceeding the size target.

The clustering phase usually results in a number of regions with size close to S centered around the “hottest” nodes of the WCFG. However, as often happens with greedy algorithms, there are usually many small one- and two-block clusters left in between. Because the presence of too many small regions is undesirable, there is a second, “correction” phase to the heuristic. In this phase, any cluster whose size is less than a certain fraction αS of the size target is merged with the neighboring cluster with which its connection is strongest, regardless of size limitations.

For the experimental evaluation presented in Chapter 6, we settled on the values $S = 500$ instructions and $\alpha = 0.1$ as a good tradeoff between optimizability and compile-time dilation, after trying several values.

5.2 Region Encapsulation

Once the WCFG has been divided into regions, the compiler must transform each region into a self-contained compilation unit. As described by Hank et al. [22], this can be achieved by annotating the virtual registers that are live-in at each region entry and live-out at each region exit. Analysis routines can then insert special CFG nodes before each entry and after each exit. These nodes will appear to “define” live-in registers and “use” live-out registers respectively. Optimization routines can then treat these special nodes conservatively. For example, if a virtual register use has reaching definitions from one of the special region entry nodes, it cannot be a candidate for constant propagation. Similarly, if a virtual register definition has an upward exposed use coming from one of the special exit nodes, then that definition will never be considered dead.

There are two challenges in applying this region encapsulation technique to PBE. The first is that the liveness of registers at region boundaries has to be calculated in a program-wide analysis pass. As program size grows, this pass can become prohibitively expensive. To alleviate this problem, the PBE compiler performs this analysis on an abbreviated *program skeleton*, instead of analyzing the entire WCFG. Since region encapsulation needs liveness information on region entries and exits, such nodes have to be present in the skeleton. Also, since the PBE analysis presented in Chapter 4 needs to match call and return arcs in order to produce accurate results, the skeleton graph must also represent call and return arcs adequately. Still, nodes inside a region that do not relate to calls, returns, or region entries and exits can be abstracted away into *skeleton edges*. Therefore, the program skeleton consists of the following elements:

- Region entries and exits.
- The original WCFG arcs connecting region exits to region entries.
- Call and return arcs, plus their source and destination nodes.
- Skeleton edges, which summarize the remaining nodes of the WCFG. Such edges begin at region entries or destinations of interprocedural arcs and end at region exits or sources of interprocedural arcs. Skeleton edges are annotated with the virtual registers that may be defined or used along the part of the WCFG they represent.

Since a skeleton edge always represents a subgraph of the WCFG that lies within the same region and does not contain calls or returns, the annotations of skeleton edges can be easily computed by applying a simple (intraprocedural) dataflow analysis pass separately within each region. After the annotations of skeleton edges have been computed, a PBE analysis pass (see Chapter 4) on the entire program skeleton can yield the liveness information for region entries and exits. Since the vast majority of WCFG nodes have been abstracted away by skeleton edges, analyzing the program skeleton is much cheaper, and therefore more scalable, than analyzing the entire WCFG.

The second challenge is unmatched call and return arcs that arise from regions selected independently of the program's procedures. PBE analysis routines, which rely on the proper matching between calls and returns, can be confused by this. To avoid this problem, a few special nodes are added to each encapsulated region's CFG. A node PE is

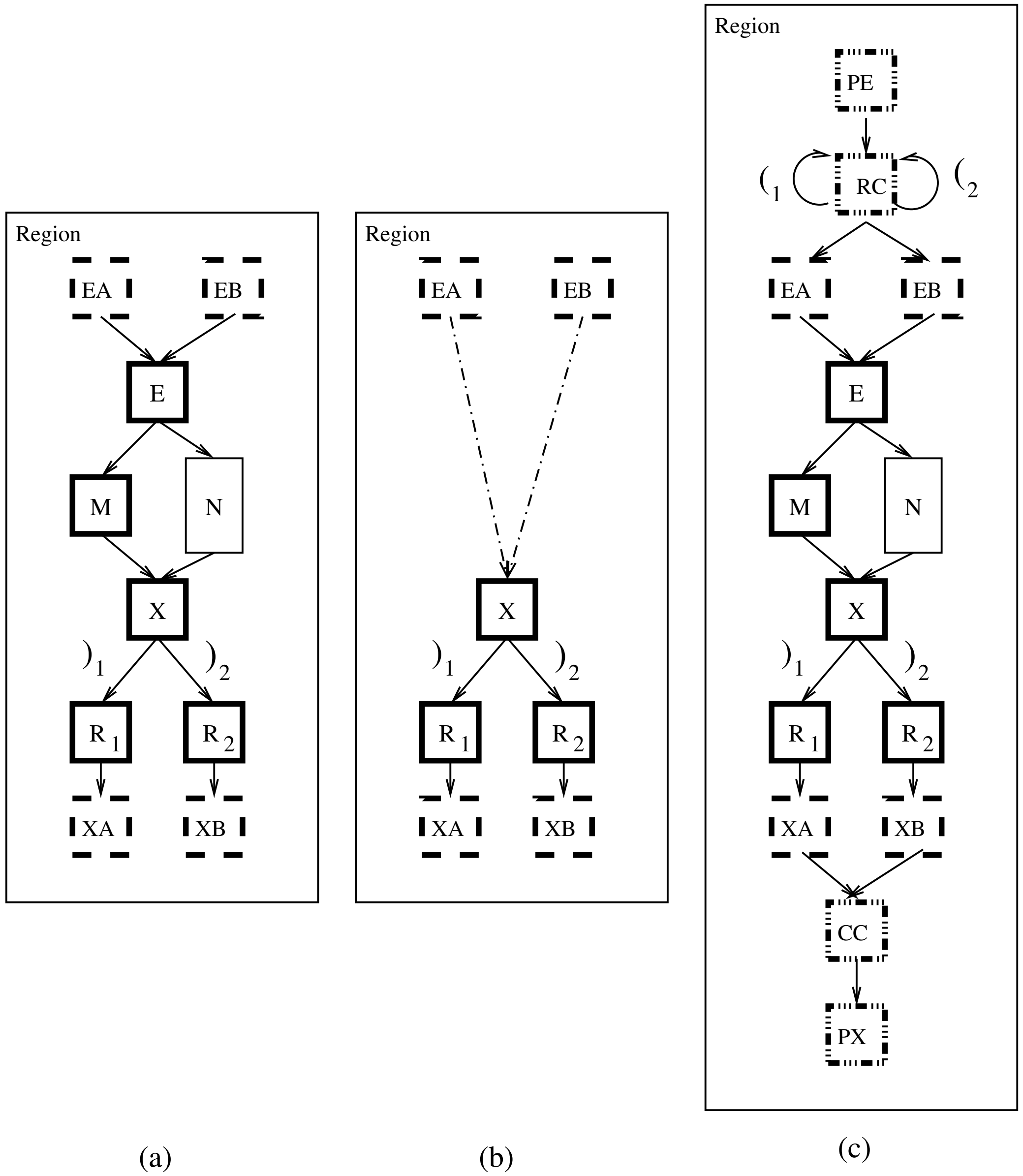


Figure 5.1: (a) A region formed around blocks E, M, N, X, R_1, and R_2 from Figure 3.2b with nodes for region entries and exits. (b) The region's program skeleton form, with blocks E, M, and N abstracted into two skeleton edges (dashed arrows). (c) The encapsulated form of the region, with program entry, program exit, call closure, and return closure nodes.

added to represent the program entry. This node is followed by a *return closure* node RC that has edges of the form $RC \xrightarrow{(_i} RC$ circling back to it, for every return annotation $)_i$ that appears in the region. In fact, RC only needs a call annotation $\xrightarrow{(_i}$ if $)_i$ appears in the region and $(_i$ appears at an edge backwards-reachable from one of the region's entries. However, enforcing this distinction makes little difference in practice. Node C is then connected to all the entries of the current region. Essentially, node C provides a matching call arc for every return arc that may be unmatched in the region. Symmetrically, the encapsulated region contains a node PX to represent the program exit, as well as a *call closure* node CC with an edge $CC \xrightarrow{)_i} CC$ circling back to it for every call annotation $(_i$ in the region.

These is illustrated in Figure 5.1. Figure 5.1a shows a region formed out of blocks E, M, N, X, R_1, and R_2 from Figure 3.2b. The region has two special entry blocks, RA and RB, to represent flows into the region, and two exit blocks, RX and RY, to represent flows out of the region. Figure 5.1b shows the skeleton form of this region. Region entries and exits are present in the skeleton, as well as blocks X, R_1, and R_2, which are sources and destinations of return arcs. Blocks E, M, and N, on the other hand, have been abstracted away through two skeleton edges, shown as dashed arrows in the figure. Finally, Figure 5.1c shows the fully encapsulated form of the region, with program entry, program exit, return closure, and call closure nodes. The return closure node has two call edges that match the two return annotations $)_1$ and $)_2$. The call closure node has no return edges, since no call annotations appear in this region.

With regions encapsulated as described above, further analysis and optimization routines do not need to consider the entire WCFG. This avoids scalability problems in later phases of the compiler, thus allowing PBE to be used on real-world applications.

Chapter 6

PBE Experimental Evaluation

In order to evaluate the ideas presented in Chapters 3 to 5, an experimental platform was developed that allows the comparison of a PBE-enabled compilation process to a process based on aggressive inlining. For that purpose, PBE was implemented in the Liberty group's research compiler, called VELOCITY. In the experiments described below, the VELOCITY compiler was combined with IMPACT [4], an existing research compiler developed at the University of Illinois at Urbana-Champaign. IMPACT is the best-performing compiler for the Intel Itanium architecture for SPEC codes, thus providing a credible experimental baseline.

Section 6.1 describes the experimental setup, based on the above two compilers, in more detail. Using this experimental setup, Section 6.2 compares PBE with aggressive inlining in terms of both code growth and generated code performance. Finally, Section 6.3 evaluates PBE's compile-time costs.

6.1 Experimental Setup

In our experimental setup IMPACT is used as a front end, compiling C benchmarks to its low-level IR, called Lcode. IMPACT also annotates Lcode with the results of aggressive alias analysis [41], which are exploited in later optimization phases. Lcode is then used as input to VELOCITY, which translates it to its own low-level IR.

In order to support the experiment described below, VELOCITY implements an aggressive inlining routine, which very closely follows the inlining routine of the IMPACT compiler. Needless to say, VELOCITY also implements the various phases of PBE: procedure unification (Section 3.3), region formation and encapsulation (Chapter 5), superblock formation (Section 3.4.1), and area code specialization (Section 3.4.2).

For later compilation phases, VELOCITY contains an aggressive classical optimizer that includes global versions of partial redundancy elimination, dead and unreachable code elimination, copy and constant propagation, reverse copy propagation, algebraic simplification, constant folding, strength reduction, and redundant load and store elimination. A local version of value numbering is also implemented. These optimization routines are applied exhaustively in a loop, until none of them can find any further transformation opportunities. To support this optimizer, VELOCITY contains a general-purpose dataflow analysis library for bit-vector problems that supports both intraprocedural analysis and the PBE analysis algorithm (Chapter 4). Finally, VELOCITY includes register allocation and superblock-based scheduling targeted to the Itanium 2 architecture. The heuristics for both these routines once again came from the corresponding routines in the IMPACT compiler.

Unfortunately, VELOCITY does not yet handle most of the advanced performance features of the Itanium architecture, such as predication, control and data speculation, and prefetching. Also, the scheduler does not take the Itanium architecture's bundling constraints into account.

Immediately after the Lcode is input into VELOCITY, it is profiled. (Although IMPACT also contains a profiler, we chose to reimplement profiling in VELOCITY, mostly due to difficulties in propagating IMPACT's profile weights accurately.) The code then undergoes a first pass of intraprocedural classical optimization. This “cleans up” the code, making subsequent inlining, region formation, and TCS heuristics more effective. From then on, each benchmark's compilation follows three different paths. The first path contains neither inlining nor PBE. The results of this path form the baseline in our measurements. In the second path, the code is subjected to an aggressive inlining pass, closely following IMPACT's inlining heuristics. The third path represents PBE. Procedures are unified, the code is divided into regions, regions are encapsulated, and an area specialization pass is applied. All three paths continue by applying superblock formation, another pass of classical optimization, register allocation, and list scheduling.

The above three paths were applied to eight benchmarks from the SPEC suites:

- `124.m88ksim` and `129.compress` from the SPECint95 benchmark suite and
- `164.gzip`, `179.art`, `181.mcf`, `183.equake`, `188.ammp`, and `256.bzip2` from the SPECint2000 benchmark suite.

Unfortunately, it was not possible to run the experiment on the entire SPECint2000 suite, due to various bugs in VELOCITY's implementation at the time.

6.2 PBE Performance Evaluation

To compare the performance of PBE versus inlining, we subjected the above eight benchmarks to the three different compilation paths described in the previous section: strict procedure-based compilation, inlining, and PBE. Runtime performance was measured by profiling or running each version of the generated code using the SPEC suite's training inputs. We chose to use the training inputs (i.e. the same inputs that are used in the compiler's profiling phase) rather than evaluation inputs (i.e. the inputs normally used by architecture manufacturers for publishing SPEC performance numbers) for this experiment in order to obtain an accurate assessment of PBE's capabilities versus inlining, without being distracted by issues of profile input suitability.

Runtime performance was measured in two different ways. Figure 6.1 measures performance by means of dynamic cycle counts. These cycle counts were obtained by profiling

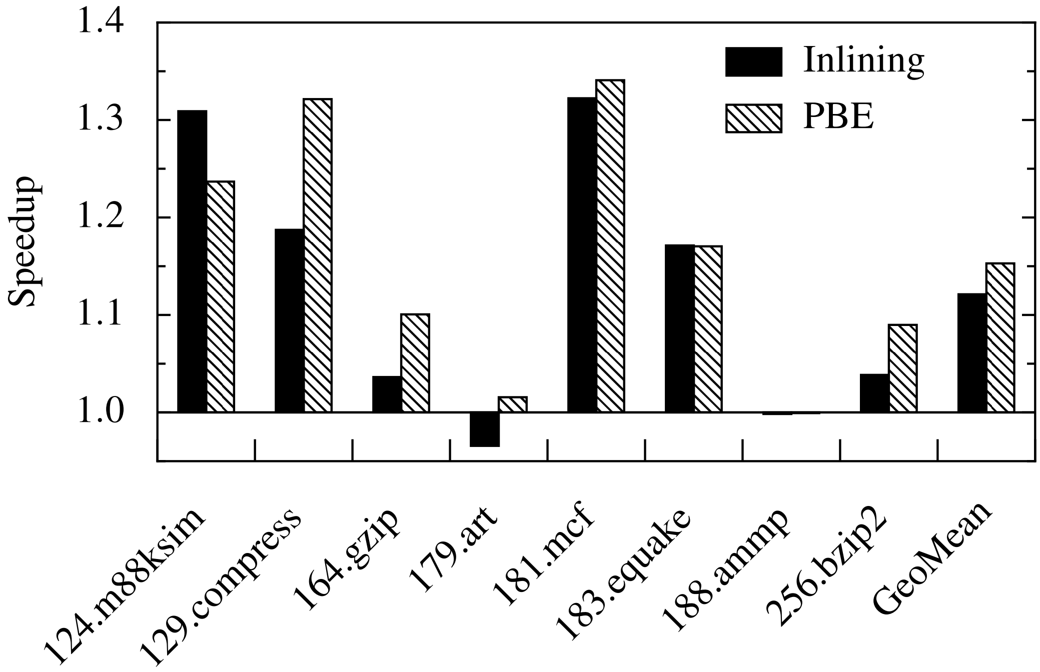


Figure 6.1: Performance benefits of inlining and PBE on training inputs over strict procedure-based compilation (dynamic cycle count).

the generated code and then tallying the “predicted cycles” of each basic block, defined as the product of the block’s schedule height and the block’s profile weight. This is equivalent to simulating the code on an Itanium-like 6-wide uniform VLIW machine with perfect cache and branch prediction behavior. Figure 6.2, on the other hand, measures the actual runtime of each executable on an unloaded HP workstation zx2000 with a 900MHz Intel Itanium 2 processor and 2Gb of memory running RedHat Advanced Workstation 2.1. Runtime numbers were collected using HP’s *pfmon* tool [15]. Finally, Figure 6.3 measures the code growth caused by inlining and PBE compared to the baseline.

As the dynamic cycle count graph (Figure 6.1) shows, PBE performance beats aggressive inlining, 15% versus 13% on average. PBE’s performance advantage is much more pronounced in certain individual cases, such as `129.compress` (32% versus 19%) and `164.gzip` (10% versus 3.6%). There is only one benchmark, `124.m88ksim`, for which PBE underperforms inlining. This happens mostly because of poor compilation unit choices by the region formation heuristic. A more sophisticated region selection method,

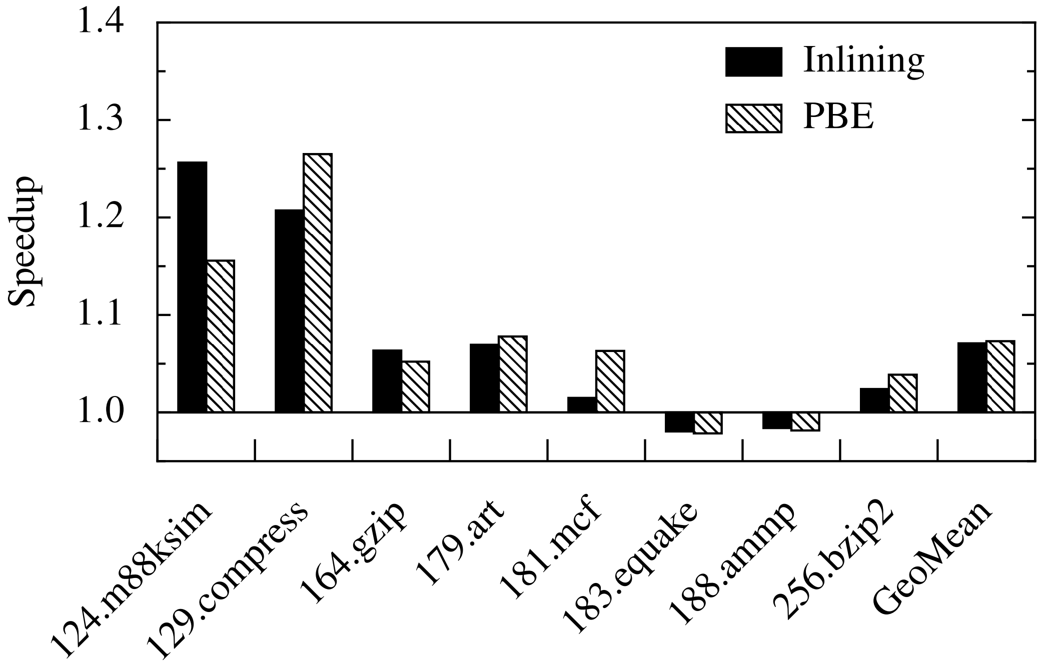


Figure 6.2: Performance benefits of inlining and PBE on training inputs over strict procedure-based compilation (runtime performance).

along the lines described in Section 10.2, would make such pitfalls even more unlikely.

Similar performance gains are shown in the runtime performance graph (Figure 6.2), though the overall speedup is about half the gain estimated from dynamic cycle count. The smaller performance differences in the second figure can be attributed to Itanium 2 runtime overheads that are not handled by VELOCITY in any of the three compilation paths, as explained in the previous section.

Most importantly, PBE achieves these performance benefits with only about half the code growth of inlining, 23% versus 45% on average (Figure 6.3). This is important for several reasons. First, smaller code can often lead to better instruction cache behavior. Second, unlike experimental compilers such as IMPACT or VELOCITY, most commercial compilers cannot tolerate code growth like that caused by this experiment's inlining routine. As a result, the inlining heuristics used in industrial compilers are much less aggressive. In such an environment the performance gap between PBE and inlining would be significantly bigger. These performance versus code growth restrictions are even more pronounced in

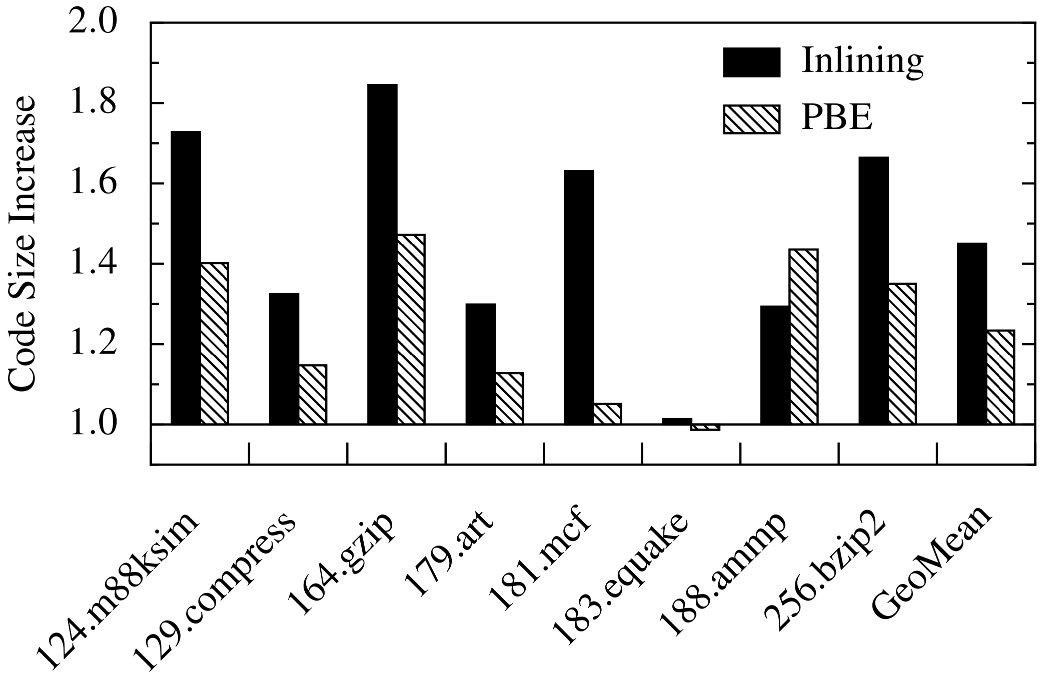


Figure 6.3: Code growth caused by inlining and PBE compared to strict procedure-based compilation

the embedded system domain, often leading compiler writers to entirely forgo inlining. In this setting, PBE may be the only way to extend optimization across procedure boundaries.

6.3 PBE Compilation Overhead

The compile-time overhead of PBE is expected to come from two sources: First, PBE starts by applying procedure unification, region formation, and region encapsulation. Traditional compilation, on the other hand, does not have to apply these phases. TCS routines that are not normally present in a non-PBE compiler, such as area specialization, also have to be added to PBE's up-front overhead. Figure 6.4 shows the percent of total compile time spent performing unification, region formation, region encapsulation, and area specialization. As seen in Figure 6.4, the initial phases of PBE consume between 2% and 5% of compile time, with a geometric mean of 3%. This shows that the techniques discussed in Chapter 3 cause only a trivial increase in compilation time. In particular, the use of an abbreviated

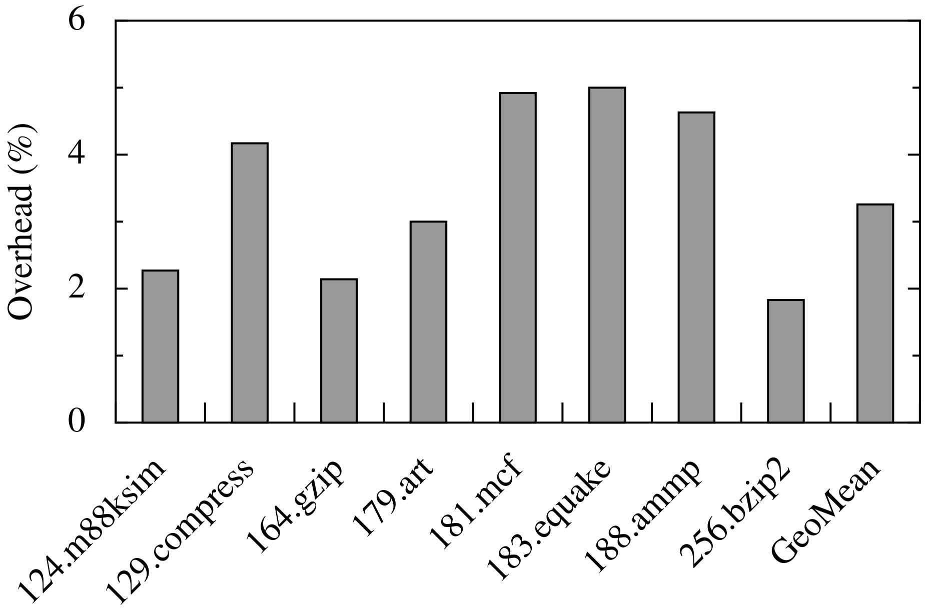


Figure 6.4: Overhead of unification, region formation, region encapsulation, and area specialization as a percentage of total compilation time.

program skeleton during region encapsulation (Section 5.2) avoids the excessive overhead that occurs when analyzing the whole program.

The second and most important source of PBE compile-time overhead comes from increased effort in later compilation phases. The full overhead of PBE, including startup costs and increased optimization time, can be seen in Figure 6.5. This figure shows that PBE does not incur prohibitive compile time costs. On average, a PBE compilation is 70% slower than an inlining compilation, which is itself twice as slow as the baseline. Since Figure 6.4 showed that the overhead of PBE's initial phases is relatively small, most of this compile-time dilation can be ascribed to extra optimization and analysis effort within regions. Partly, this variance in compile times (especially the extreme cases, such as `188.ammp`), is due to the experimental nature of our compiler. Like most experimental compilers, VELOCITY performs exhaustive dataflow optimization and has to run several dataflow analysis routines on entire procedures or regions before every optimization routine. Most commercial compilers do not follow this approach because it leads to compile-time volatility. Compile time limiting techniques common in industrial-strength compilers, such as restricting most

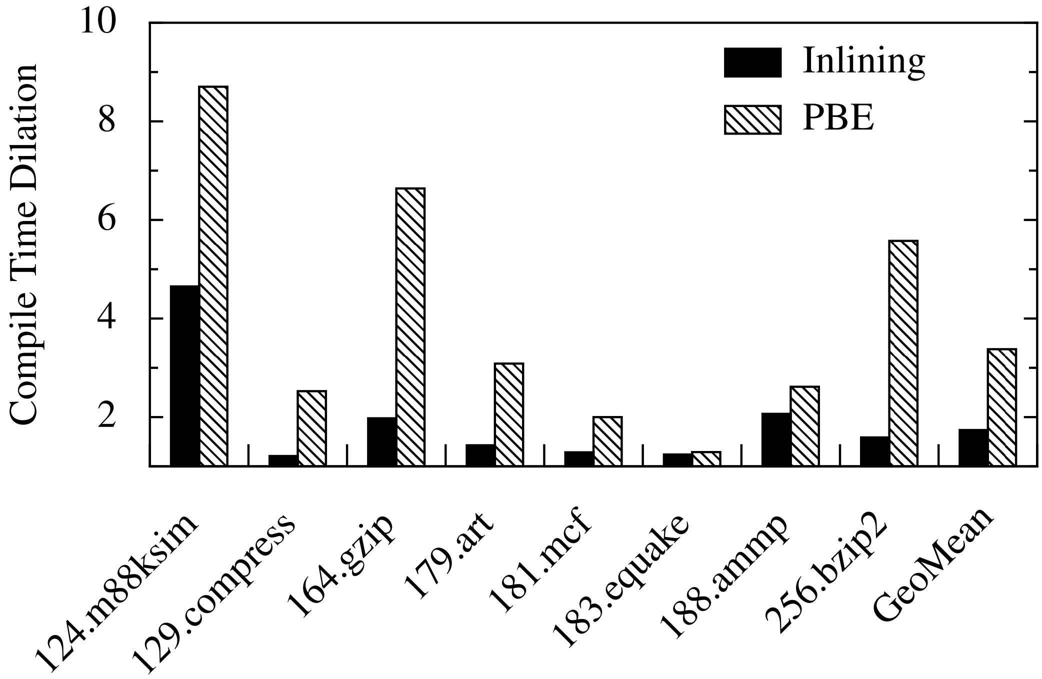


Figure 6.5: Compile time dilation for inlining and PBE compared to strict procedure-based compilation: each bar represents the ratio of the compile time consumed by inlining and PBE respectively over the compile time consumed by the baseline compiler.

classical optimizations to within basic blocks, capping the number of optimization passes applied, or tolerating slightly inaccurate dataflow information, could reduce the compile-time gap further. However, such an approach would not allow us to evaluate the full performance impact of either inlining or PBE, and thus would not be appropriate for a research experiment.

Chapter 7

The Optimization Decision Problem

While the first part of this dissertation dealt with the problem of optimization scope, the second part deals with the performance limitations introduced by the compiler's difficulty in reaching the correct optimization decisions. Since modern processors often contain nonuniform resources, explicit parallelism, multilevel memory hierarchies, speculation support, and other advanced performance features, judging the impact of a code transformation on final code quality cannot be achieved by simple metrics such as instruction count or code size. Instead, an optimizing compiler has to carefully balance a set of performance factors, such as dependence height, register pressure, and resource utilization. More importantly, the compiler must also anticipate dynamic effects, such as cache misses and branch mispredictions, and avoid or mask them if possible.

To accomplish this task, a compiler needs a large number of complex transformations. Unlike traditional compiler optimizations, such as dead code elimination or constant folding, few of these transformations are universally beneficial. Most of them constitute tradeoffs, improving some performance factors while downgrading others. For example, loop unrolling increases instruction-level parallelism (ILP) but may adversely affect cache performance, whereas if-conversion avoids branch stalls but increases the number of instructions that must be fetched and issued. Worse, the final outcome of any code transformation ultimately depends on its interactions with subsequent transformations. For example, a software pipelining transformation that originally seems beneficial may lead to more spills during register allocation, thus worsening performance.

Clearly, a successful optimizing compiler must not only incorporate a rich set of optimization routines, but must also correctly determine where and when to apply each one of them. In today's compilers, this is usually achieved through the use of *predictive heuristics*. Such heuristics examine a code segment right before an optimization routine is applied on it, and try to *a priori* judge the optimization's impact on final performance. Usually, a great amount of time and effort is devoted to crafting accurate heuristics. However, a heuristic's task is complicated not only by the complexity of the target platform, but also by the fact that it must anticipate the effect of the current code transformation on all subsequent optimization passes. To make a successful prediction in all cases, each heuristic would ultimately have to be aware of all other heuristics and optimization routines, and all the ways they might interact. Furthermore, all heuristics would have to be changed every time an optimization routine is added or modified. In today's complex optimizers this is clearly an unmanageable task. Therefore it is to be expected that real-world predictive heuristics will make wrong optimization decisions in many cases.

To manage these complications, compiler writers do not fully specify the heuristic and optimization behavior during compiler development. Instead, they leave several optimization *parameters* open. For example, the maximum unroll factor that a loop unrolling heuristic may use can be such a parameter. Similarly, an if-conversion parameter may control exactly how balanced a branch has to be before if-conversion is considered. The values of such parameters are determined during a tuning phase, which attempts to maximize a compiler's performance over a representative sample of applications. In essence, such parameters give the compiler's components a limited ability to automatically adapt to the target architecture, to the target application set, and to each other.

Parameterization and tuning have proven to be very effective in improving a modern compiler's performance. However, they are still an imperfect answer to modern optimization needs. No matter how sophisticated a tuning process is, the end result is still a single, rigid compiler configuration, which then has to be applied to code segments with widely varying optimization needs. In the end, tuning can only maximize the *average* performance across the sample applications. However, this "one size fits all" approach will unavoidably sacrifice optimization opportunities in many individual cases. This effect is especially pronounced when the compiler is applied on code that is not well represented in its tuning sample. Section 7.1 elaborates further on the problems caused by this compilation approach.

To address these limitations of traditional compiler organization, *iterative compilation* has been proposed [1, 13, 16, 29, 32, 40, 56, 57]. Instead of relying on *a priori* predictions, an iterative compiler applies many different optimization configurations on each code segment. It subsequently compares the different optimized versions of each segment and decides which one is best *a posteriori*. This allows the compiler to adapt to the optimization needs of each code segment. Previous research indicates that iterative compilation can provide significant performance benefits.

The problem with current iterative compilation approaches is their brute force nature. Such approaches identify the correct optimization path by considering all, or at least a great number of, possible optimization paths. This incurs prohibitive compile time costs. Therefore, iterative compilation has been currently limited to small parts of the optimization process, small applications, and/or application domains where large compile times are acceptable, such as embedded processors and supercomputing. Section 7.2 discusses a representative sample of the iterative compilation bibliography.

7.1 Predictive Heuristics and Their Problems

Typically, modern compilers apply a “one size fits all” approach to optimization, whereby every code segment is subjected to a single, uniform optimization sequence. The only opportunity to customize the optimization sequence to the needs of each individual code segment is offered through predictive heuristics. However, the difficulty of characterizing interactions between optimization routines, as well as the complexity of the target architecture, make it very hard to make accurate optimization decisions *a priori*. As a result, no single compiler configuration allows optimizations to live up to their maximum potential. Although such an optimization process can be tuned for maximum average performance, it will still sacrifice important optimization opportunities in individual cases. Section 7.1.1 strengthens this argument through a bibliographic survey. Section 7.1.2 presents an experimental setup based on the Intel Electron compiler, which will be used both in this chapter and in the experimental evaluation of Chapter 9. Section 7.1.3 provides a more quantitative argument through an experimental evaluation of heuristic shortcomings in the Intel Electron compiler. Finally, Section 7.1.4 briefly surveys the behavior of predictive heuristics in other compilers.

7.1.1 Predictive Heuristics in Traditional Compilation

Evaluating the full range of predictive heuristic design and use is beyond the scope of this dissertation. Instead, this section provides a sample of predictive heuristic work, in order to strengthen the argument that the art of *a priori* predicting the behavior and interactions of compiler optimizations is at best an inexact science.

Whitfield et al. [55] propose an experimental framework for characterizing optimization interactions, both analytically and experimentally. This allows a compiler developer to examine different ways to organize optimization routines and study how they interact. Studies performed using this framework underscore the fact that optimization routines interact heavily and in ways that are difficult to predict. Ultimately, no single optimization organization is ideal for all cases; each compiler configuration exhibits different strengths and weaknesses, making it well-suited for only a fraction of targeted codes.

Recognizing the difficulty in hand-crafting heuristics, Stephenson et al. [48] let heuristics "evolve" using genetic algorithms. When the genetic algorithm is used to identify the best overall register allocation heuristic for a given benchmark set, it results in a register allocation routine offering a 9% performance improvement. However, when the genetic algorithm is allowed to converge on a different heuristic for each benchmark, the performance improvement is 23% on average. Thus this work underscores the fact that a "one size fits all" optimization approach, even a well-tuned one, is liable to sacrifice performance.

Other work has focused on addressing particularly problematic optimization interactions and developing better heuristics to circumvent performance pitfalls. Heuristics that try to avoid register spilling due to overly aggressive software pipelining have been proposed [19, 35]. Although the proposed heuristics are quite sophisticated, the authors describe cases that the heuristics cannot capture. Among the best studied optimization interferences are those that occur between scheduling and register allocation. Proposed heuristic techniques seek to minimize harmful interferences by considering these two code transformations in a unified way [8, 17, 33, 38]. Continuing efforts in this area indicate that none of the existing heuristics can fully capture these interferences.

Hyperblock formation and corresponding heuristics have been proposed to determine when and how to predicate code [36]. However, even with these techniques, the resulting predicated code often performs worse than it did originally. In an effort to mitigate this problem, techniques that partially reverse predication by reinserting control flow have been proposed [5]. This need to reexamine and roll back code transformations underscores the difficulty of making *a priori* optimization decisions.

These works are just a sample of the research done to address problems of predictive heuristics. The continuing effort to design better predictive heuristics and to improve compiler tuning techniques indicates that the problem of determining if, when, and how to apply optimizations is far from solved.

7.1.2 Experimental Setup

The previous section argued that any traditional compiler, no matter how well-tuned, is bound to sacrifice performance opportunities due to incorrect optimization decisions. In order to quantify this performance loss, we used an experimental setup based on the Intel C and C++ compiler for the Itanium Processor Family (IPF), also known as Electron. Since Electron is the SPEC reference compiler for IPF, it provides a credible experimentation base. Also, IPF is an especially interesting target architecture, since its explicit parallelism and its complicated performance features make the proper application of aggressive optimizations crucial to achieving good performance. For our experimental baseline we used Electron version 6.0 invoked with the command-line parameters `-O2 -ip -prof_use`, which enable intraprocedural optimization, interprocedural optimization and analysis within the same source-code file, and profile information use. This is very close to the compiler configuration used to report the official SPEC numbers for Itanium. (The exact SPEC reference command line is `-O2 -ipo -prof_use`, which also enables cross-file interprocedural optimization and analysis. However, using these settings in conjunction with our experimental harness proved impossible). Note that the `-O2` option is used instead of the more aggressive `-O3`, both for the official SPEC measurements and for our baseline. This is because the more aggressive optimization settings enabled by `-O3` often cause significant performance degradation instead of improvement, especially for nonscientific codes. This makes a study of Electron’s optimization decision failures all the more interesting.

For this study, the behavior of several Electron optimizations was varied and the impact of these variations on compiled code performance was observed. The full list of optimization parameters studied is given in Table 7.1. The different variations of Electron were

#	Parameter	Meaning
1	Optimization level	`-O2` (default) performs standard optimizations, including loop optimizations [26]. `-O3` performs all `-O2` optimizations plus riskier optimizations that may degrade performance, including aggressive loop transformations, data prefetching, and scalar replacement.
2	HLO level	Same as above, but affects only the high-level optimizer.
3	Microarchitecture type	Optimize for Itanium (default) or Itanium 2. Affects the aggressiveness of many optimizations.
4	Load/store coalescing	On by default. Forms single instructions out of adjacent loads and stores.
5	If-conversion	On by default.
6	Nonstandard predication	Off by default. Enables predication for if blocks without else clauses.
7	Software pipelining	On by default.
8	Software pipeline outer loops	Off by default.
9	Software pipelining if-conversion heuristic	On by default. Uses a heuristic to determine whether to if-convert a hammock in a loop that is being software pipelined. If disabled, every hammock in the loop is if-converted.
10	Software pipeline loops with early exits	On by default.
11	HLO after loop normalization	Off by default. Forces HLO to occur before loops are normalized, effectively disabling some optimizations.
12	Loop unroll limit	Maximum loop unrolling factor. Values tried: 0, 2, 4, 12 (default).
13	Update dependences after unrolling	On by default. If disabled, it effectively limits optimization aggressiveness on unrolled loops.
14	Prescheduling	On by default. Runs scheduling before *and* after register allocation. If disabled, runs scheduling only after register allocation.
15	Scheduler ready criterion	Percentage of execution-ready paths an instruction must be on to be considered for scheduling. Values tried: 10%, 15% (default), 30%, and 50%.

Table 7.1: Electron optimization parameters used in the experiments of Chapters 7 and 9.

applied to the following benchmarks:

- The SPECint2000 benchmarks `164.gzip`, `175.vpr`, `176.gcc`, `181.mcf`, `186.crafty`, `197.parser`, `253.perlbmk`, `254.gap`, `255.vortex`, `256.bzip2`, and `300.twolf`.
- the SPECcfp2000 benchmarks `177.mesa`, `179.art`, `183.equake`, and `188.ammp`.
- the SPECint95 benchmarks `099.go`, `124.m88ksim`, `129.compress`, and `132.ijpeg`.
- the MediaBench benchmarks `adpcmdec`, `adpcmenc`, `epicdec`, `epicenc`, `g721dec`, `g721enc`, `jpegdec`, and `jpegenc`.
- the parser generator `yacc`

Note that `252.eon` is missing from the SPECint2000 benchmarks above. This is because our experimental support programs could not handle `C++`. More benchmarks are missing from the SPEC95 and MediaBench suites. Quite a few of these benchmarks did not compile or run on Itanium since they were written for 32-bit architectures. For others, the compilation process failed for certain configurations tried. This is not surprising, since this experiment exercises parts of Electron's optimizer that were not enabled in the officially released code, and thus were probably not fully debugged.

For the performance measurements, executables were run on unloaded HP i2000 Itanium workstations running Red Hat Linux with kernel version 2.4.18. Execution times were obtained using the performance counters of the IPF architecture with the help of the `libpfm` library [43]. Because most of our experiments required measuring the computation cycles consumed by each procedure in a benchmark, we developed an instrumentation system that directly manipulates the assembly language produced by Electron, adding appropriate actions at each procedure entry and exit. These actions involve reading and resetting IPF's performance counters and accumulating each procedure's cycle count in special memory locations. These cycle counts are printed to a file by an exit function, installed through `atexit()`. Whenever whole program runtimes are reported, these are taken to be the sum of the cycles spent in a program's source procedures, excluding time spent in system calls and precompiled libraries. Reported cycle counts do not contain cycles spent in the instrumentation system itself. Also, since the instrumentation system works directly on assembly language, it does not disturb any part of the optimization process. Some cache interference is unavoidable, but it is limited to a few bytes of data accessed per function entry or exit. Each benchmark was run enough times to reduce random drift in the measurements to below 0.5%. The times that a benchmark had to be run varied according to the characteristics of the benchmark and its input sets, from 3 for the bigger benchmarks to about 20 for the smaller ones.

7.1.3 Heuristic Failure Rates

To quantify the frequency of predictive heuristic failure in Electron, the following experiment was performed. The optimization parameters appearing in Table 7.1 were grouped into six categories: overall optimization approach (1st to 3rd parameter), load/store handling (4th parameter), predication (5th and 6th parameters), software pipelining (7th to 10th parameters), other loop optimizations (11th to 13th parameters), and scheduling (14th and 15th parameters). For each one of these categories we determined how often a nonstandard setting of the category's parameters produces noticeably faster code than the default setting. For this purpose we tried each possible parameter setting for each category on a group of code samples comprising the most important procedures in our benchmark set, namely all procedures that consume at least 10% of their benchmark's runtime. There are 66 such functions in our benchmark set.

The results of this experiment can be seen in Figure 7.1. For each category, this graph shows how often a parameter setting other than the default results in at least 3%, 5%, 10%,

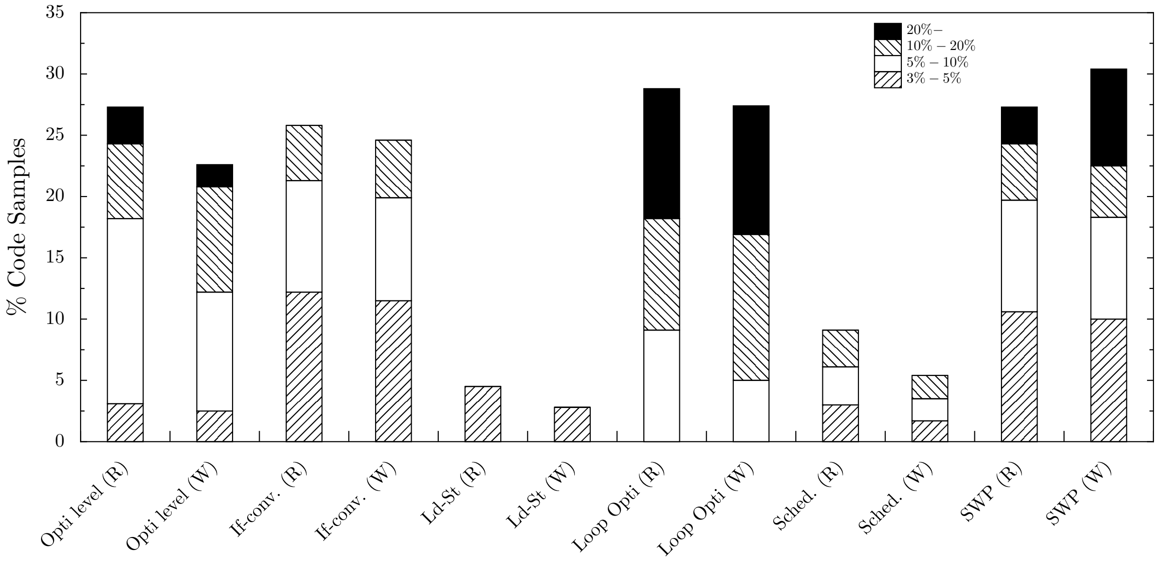


Figure 7.1: Percentage of procedures for which a setting other than the default of a category's parameters causes a speedup greater than 3%, 5%, 10%, and 20% over Electron's default configuration.

and 20% better performance than Electron's default setting for the category's parameters. For columns marked with "R" all procedures count the same, whereas for columns marked with "W" procedures are weighed by their execution weight in the corresponding benchmark.

As we can see in Figure 7.1, the default setting in each category performs well in a majority of cases. However, a significant number of procedures is not served well by the compiler's default configuration. For example, one of every four procedures could improve its performance by at least 5% if the overall optimization approach were set to different parameters. Similarly, one out of every five procedures would be at least 10% faster if the loop optimizations were customized to its needs. These results provide evidence that even a well-tuned single-configuration compilation process cannot fit all codes, thus leaving significant performance benefits unrealized.

7.1.4 Heuristics in Other Compilers

The failure of heuristic-driven compilation to make correct optimization decisions is not a phenomenon peculiar to Electron. In one small experiment, we varied the loop unrolling factor used by the IMPACT compiler [4] incrementally from 2 to 64. The benchmark `132.ijpeg` performed best for the default loop unrolling factor of 2. However, a performance increase of 8.81% was achieved by allowing each function in `132.ijpeg` to be compiled with a different loop unrolling factor. In a bigger experiment involving 72 different configurations, the individually best configurations for `130.li` and `008.espresso` achieved 5.31% and 11.74% improvement over the globally best configuration respectively.

As noted in Section 7.1.1, the experimental compiler used in [48], which is based on Trimaran, exhibits similar behavior. Allowing the register allocation strategy to be customized on a benchmark by benchmark basis leads to a 13% performance gain over the globally best register allocation strategy.

The experiment described in [2] focuses on GCC targeting Pentium architectures. Reported results show a performance improvement of up to 6% if the compiler configuration is customized on a per-program basis.

7.2 Iterative Compilation

As previously discussed, traditional optimizers subject every code segment to a single, uniform optimization process. Iterative compilation takes a different approach. Instead of relying on *a priori* predictions, an iterative compiler applies many different optimization sequences on each code segment. The different optimized versions of each code segment are then compared using an objective function, and the best one is output. Thus iterative compilation is able to find the “custom” optimization approach that best meets each code segment’s needs. Although this approach usually results in significant performance gains, it requires prohibitively large compile times. This has prevented iterative compilation from being broadly applied. Most importantly, this has rendered iterative compilation unsuitable for production compilers.

Several iterative compilation approaches have been proposed in the past. Section 7.2.1 presents approaches that focus on dataflow optimization routines and their phase orders. Works presented in Section 7.2.2 focus more on loop optimizations for small loop kernels on embedded systems. In an interesting variant to iterative compilation, methods presented in Section 7.2.3 perform an iterative exploration on an abstraction of the code before the actual optimization happens.

7.2.1 Iterative Classical Optimization

Cooper et al. [13] propose a compilation framework called *adaptive compilation*, which explores different optimization phase orders at compile time. The results of each phase order are evaluated *a posteriori* using one of several objective functions. This system is experimentally evaluated on a small FORTRAN benchmark. Depending on the objective function selected, the adaptive compilation system can produce a 13% reduction in code size or a 20% reduction in runtime relative to a well-tuned traditional compiler. Although some rudimentary pruning techniques are used, the system still needs from 75 to 180 trials before it can identify a solution within 15% of the ideal one.

Instead of traversing possible phase orders in a brute-force manner, Kulkarni et al. [32] propose a selective exploration of phase orders using a genetic algorithm. In its pure form, this approach needs to compile and execute each procedure 2000 times. The benefits of this exploration can be a reduction in the resulting code's runtime and code size by up to 12% each, with respective averages of 3% and 7%. When applying several search pruning techniques proposed by the authors, the same benefit can be achieved with about 70% less compile-time dilation. Although not astronomical, this level of compile-time dilation is obviously too steep for general purpose compilers. For this reason, this optimization method is targeted towards the embedded systems domain, and it is experimentally evaluated on relatively small benchmarks.

7.2.2 Iterative Embedded Loop Kernel Optimization

The OCEANS compiler group [1] has also investigated iterative compilation approaches, mainly within the context of embedded system applications. An initial study [29] on iterative applications of loop unrolling and tiling on three small numerical kernels proves that the proposed approach can cause up to a fourfold increase in generated code performance. A more realistic study [16], involving three loop transformations applied to more sizable numerical benchmarks, achieves a 10% improvement over an aggressively optimizing traditional compiler. Despite the presence of pruning techniques, the system still needs to apply up to 200 different optimization sequences before this performance gain is achieved.

The GAPS compiler project [40] studies the iterative application of loop transformations on numeric benchmarks for parallel processors. Genetic algorithms are used to guide the search for the best optimization sequence at compile time. When applied on a numeric benchmark, the GAPS compiler is able to produce a 75% performance improvement in comparison to the native FORTRAN compiler. The compile time needed for a single small benchmark is about 24 hours.

7.2.3 Iterative Predictive Heuristic Methods

In an interesting variant of iterative compilation, Wolf et al. [56] and Zhao et al. [57] present algorithms for combining high-level loop transformations, such as fusion, fission, unrolling, interchanging, and tiling. For each set of nested loops the proposed algorithms consider various promising combinations of these transformations. Instead of fully applying each transformation sequence, the algorithms preevaluate them by “simulating” their application using a skeleton of the original code. A performance estimator then selects a single sequence for actual application. When evaluated on scientific code, [56] reports a 15% performance improvement over noniterative approaches. On the other hand, [57] reports much larger benefits (up to 120%), but only when using a trivial baseline that applies the aforementioned optimizations indiscriminately. Although no compile-time results are included in either paper, the proposed algorithms seem reasonably efficient. However, these algorithms cannot be generalized to other optimizing transformations or to nonnumerical applications, since they depend on a thorough understanding of the interactions between these particular transformations within the numerical application domain.

Chapter 8

Optimization-Space Exploration

To overcome the optimization decision limitations presented in Chapter 7, this chapter presents Optimization-Space Exploration (OSE), a novel iterative compilation method. OSE realizes the performance potential of iterative compilation while addressing the applicability limitations of existing approaches. This makes OSE the first iterative compilation method suitable for general-purpose, industrial-strength compilers.

Like other iterative compilation approaches, OSE applies different optimization configurations to each code segment. The final decision about which optimization configuration performs best is taken *a posteriori*, that is after the resulting optimized versions of the code have been examined. However, OSE differs from other iterative compilation approaches in several crucial ways that allow it to keep compile-time costs in check.

- Although predictive heuristics are unable to anticipate the full impact of an optimization routine on final code quality, they still encode valuable information on an optimization's behavior. Using this information, an iterative compiler can make intelligent choices as to which part of the optimization space to explore, reducing the number of different optimization configurations that have to be tried.

- For any given application set, a sizable part of the configuration space causes only modest performance gains. Thus the configuration space can be aggressively pruned during a compiler tuning phase.

- On any given code segment, the performance of different configurations is often correlated. This allows the compiler to utilize feedback to further prune the exploration space at compile time.

- Instead of selecting the best optimized version of the code by measuring actual runtimes, an OSE compiler relies on a static performance estimator. Although this approach is less accurate, it is much faster and more practical.

The remainder of this chapter examines each of the above ideas in greater detail (Sections 8.1 to 8.4). The chapter concludes with a brief discussion of OSE in the context of dynamic optimization and managed runtime environments (Section 8.5).

8.1 Limiting the Configuration Pool through Predictive Heuristics

As previously noted, predictive heuristics are unable to anticipate the full impact of a code transformation on final code quality. However, well-crafted heuristics still encode valuable information about a code transformation's behavior. OSE takes advantage of this fact in order to limit the number of different optimization configurations that need to be explored.

Consider loop unrolling as an example. For every loop, an iterative compiler would have to try a great number of different loop unrolling factors. OSE takes a different approach. A well-crafted and well-tuned loop unrolling heuristic, like the one found in a high-performance traditional compiler, is expected to identify the correct loop unrolling factor for a fair number of cases. To capture the remaining cases, configurations containing different variants of the original heuristic can be applied. For example, some such variants could restrict the maximum loop unrolling factor allowed. Configurations that forgo loop unrolling entirely can also be tried.

By trying many variants of each optimization's heuristic, OSE correctly captures the optimization needs of many more code segments than a traditional compiler. Of course, since heuristics are imperfect, this approach cannot capture *all* cases, like an exhaustive iterative compiler can. This, however, is a worthy tradeoff when considering compile-time savings. For example, the iterative compiler proposed in [16] has to consider 16 different unroll factors for each code segment. In comparison, the OSE prototype presented in Chapter 9 only considers 4 different loop unrolling heuristics. For every optimization, exploiting predictive heuristics causes a similar reduction in the number of choices under consideration. This leads to an overall reduction of the configuration space by a factor exponential in the number of optimization phases.

In subsequent discussions, any variable that controls the behavior of a heuristic or an optimization routine will be referred to as an optimization *parameter*. A full assignment of values to all parameters in an optimizing compiler forms an optimization *configuration*. The set of all configurations that an OSE compiler has to explore constitutes the *exploration space*.

8.2 Static Configuration Selection

By exploiting heuristics, an OSE compiler has to explore a smaller configuration space than that of an exhaustive iterative compiler. However, the size of this space is still prohibitively large. Since every parameter can take at least two values, the total size of the exploration space is exponential with regard to the number of available parameters. Clearly, exploring this space in its entirety at compile time would be impractical. Therefore a radically reduced configuration set has to be selected statically, that is during the OSE compiler's tuning.

Static configuration selection exploits the fact that all configurations are not equally valuable. Certain configurations may perform badly in the vast majority of cases. For example, such would be the case of configurations that inline over-aggressively on systems with small instruction caches. Such configurations can be omitted with little performance loss. Other configurations may form clusters that perform similarly in most cases. For example, on a processor with limited computational resources and small branch misprediction penalties, configurations differing only on if-conversion parameters would fall under this category. In such cases, keeping only one representative configuration from each cluster and pruning the rest would not lead to significant loss of optimization opportunities.

More formally, the goal of the static pruning process is to limit the configuration space to a maximum of K configurations with as little performance loss as possible. The performance of a configuration set is judged by applying it to a set of representative code samples S. The exact value of K is dictated by compile time constraints.

Ideally, the static selection algorithm would determine the best configuration space of size K by considering all possible combinations of K configurations. However, typical exploration spaces are so big that a full consideration of them is impractical, even during compiler tuning. For example, the full exploration space of the OSE prototype described in Chapter 9 contains 2^{17} configurations. On the machines used in the experiments of Chapter 9, the full traversal of this space would take roughly 45 years. Therefore, the static selection algorithm has to rely on a partial traversal of the configuration space, even though this may lead to suboptimal results.

The OSE static selection algorithm consists of an *expansion* step and a *selection* step, repeatedly applied until new steps do not provide significant new benefits, or until a time limit is reached.

Beginning with a set of configurations C_S, used as seeds, the expansion step constructs the set C_E of all configurations differing from one of the seeds in only one parameter. Subsequently, every configuration in C_E is applied on every code sample in S, and the runtimes of the optimized codes thus produced are measured.

The selection step then determines the K-element subset of C_E that maximizes the performance of OSE. For that purpose, the “exploration performance” of each such subset is determined, as follows: Let $R(s, c)$ be the runtime of a code sample s when optimized using an optimization configuration c. Then the exploration value of a set of configurations C on a set of code samples S is given by the formula:

$$EV(C, S) = \sqrt[|S|]{\prod_{s \in S} \min_{c \in C} R(s, c)}$$

That is, we calculate the geometric mean of the best-performing version of each code sample produced by a configuration in C. The selection step simply determines the exploration value of all K-element subsets of C_E, and selects the one with the best exploration performance. The configurations in the set thus selected become the new seeds, on which the expansion step is applied again, and so forth.

The effectiveness of the above process depends greatly on the choice of the initial seeds. A bad choice of seeds may lead to slower convergence, trap the algorithm in local minima, or both. It is therefore important to start with a configuration that is known to perform well on average. Such a configuration would roughly correspond to the optimization process of a well-tuned noniterative compiler.

8.3 Feedback-Directed Compile-Time Pruning

By exploiting feedback, an OSE compiler can dynamically prune the exploration space at compile time. On any given code segment the compiler can begin by applying a small set of configurations. Feedback on how these initial configurations performed can help the compiler make an informed choice on which configurations to try next. Feedback from these configurations can be used to select the next set to be tried, and so on. Thus only a portion of the configuration space needs to be explored for each code segment at compile time.

This approach works because different configurations are generally *correlated*. In

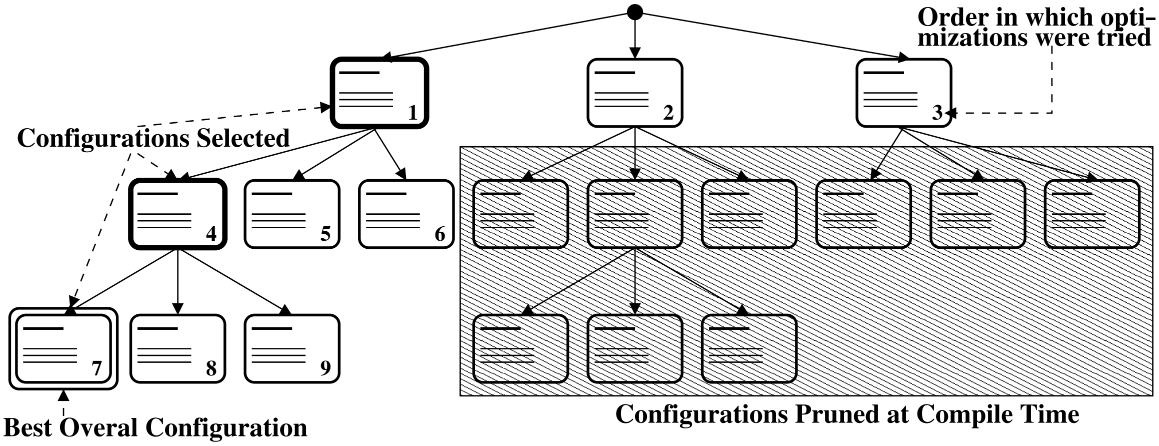

Order in which optimizations were tried
Configurations Selected
1
2
3
4
5
6
7
8
9
Best Overal Configuration
Configurations Pruned at Compile Time

Figure 8.1: Automatically generated search tree annotated based on a hypothetical run of OSE.

general, a given configuration performs well on code segments that exhibit certain *code properties*. In many cases, the code properties required by two different configurations may overlap, whereas in other cases they may be unrelated. For example, a configuration that emphasizes software pipelining will perform well on code segments containing small, straight-line loops with complicated dependence chains. On the same code segments, a configuration applying loop unrolling is also likely to perform well. On the other hand, a configuration that forgoes loop optimizations is likely to underperform. Consequently, if an OSE compiler finds out that software pipelining performs well on a code segment, it can decide to try loop unrolling configurations, while forgoing configurations that do not concentrate on loops.

The OSE compiler can exploit configuration correlations by organizing the set of K configurations, as determined in the static selection phase, into a tree, as shown in Figure 8.1. For each code segment, configurations at the top level of the tree are tried first, and the best one is selected. Subsequently, the children of the selected configuration are tried. After the best of these configurations is selected, its children are in turn tried, and so on. After the bottom of the tree is reached, the best one of the configurations selected at each level is the one that prevails.

This configuration tree has to be constructed during the tuning phase of an OSE compiler. Of course, the notion of code properties is too abstract to be practically useful in this task. However, correlations between configurations can be determined experimentally. Let us assume that an L-way tree is desired. From the K configurations remaining after static pruning, the best-performing combination of L configurations is selected as the top level of the tree. Next the set S of code samples can be partitioned into L subsets, $S_1, S_2, \ldots, S_L$. Subset S_i contains the code segments for which the i'th configuration, c_i, outperforms the other top-level configurations. For each i, the L most valuable configurations for the limited code sample set S_i can then be determined. Essentially, these are the configurations that are most likely to succeed on code segments that respond well to c_i. Therefore, these configurations become the children of c_i in the tree. Subsequent tree levels can be formed by repeating the same process on each set of siblings.

8.4 Performance Estimation

Ideally, an OSE compiler would select the best-performing version of each code segment by measuring actual runtimes. Since code segments cannot be run in isolation, the whole program would have to be compiled before the performance of a single version of a single code segment could be evaluated. Furthermore, the performance of each code segment is dependent not only on its own features, but also on the features of other code segments in the program. This is due, among other things, to cache and branch prediction effects. Therefore, an absolutely accurate judgment on a code segment's performance would have to be obtained through running it in conjunction with every other possible combination of optimized versions of all other code segments in the program. This approach is clearly impractical.

Instead, an OSE compiler makes performance judgments using a static performance estimator. Such an estimator can make predictions based on a simplified machine model and on profile data. In general, obtaining a static prediction of a code segment's runtime performance is a non-trivial task [34]. However, the job of an OSE performance estimator is much simpler, because it only needs to provide a *relative* performance prediction. Rather than trying to determine the exact runtime of a code segment, this estimator has to compare two code segments and predict which one is faster. Moreover, since the code segments compared will actually be differently optimized versions of the same source code, they will be generally similar, differing in only one or two crucial ways. If, for example, the estimator has to choose between two differently unrolled versions of the same original loop, one of the two versions will have a better schedule, whereas the other will have a smaller code size, with all their other features being very similar. Thus the estimator's task will come down to weighing the scheduling gains versus the code size expansion. In this way the OSE estimator is able to make mostly accurate predictions by simply scoring different code segments according to several performance indicators, such as static cycle count, code size, and memory access patterns. Of course, the exact form of these performance indicators and their relative weight depends on the target architecture and the target application domain. A concrete OSE performance estimator for the Itanium architecture will be presented in Section 9.1.2.

8.5 Dynamic OSE

As presented up to now, OSE is primarily a static compilation method. However, there are obvious ways in which OSE could be implemented in, and benefit from, dynamic optimization platforms (DOPs) and managed-runtime environments (MRTEs). Indeed, such environments would make OSE more successful and accurate by reducing its reliance on profile data and by eliminating the inaccuracies inherent in static performance estimation.

To respect the generally tighter timing constraints of MRTE compilation, an OSE implementation on such a system would first compile all procedures in the traditional way. Using a lightweight instrumentation, like the one described in Section 7.1.2, the system could then accumulate the execution time spent on each procedure. Once a procedure's accumulated runtime exceeds a predetermined limit, the system would judge that this procedure is “hot”, and thus worthy of further optimization. It would then proceed to produce differently optimized versions of the procedure using the configurations in the first level of the compile-time tree (Section 8.3). Calls to the original procedure would be redirected to a simple harness, which would satisfy each call by randomly invoking one of these versions. After some time, the system will have gathered enough execution time statistics to know which of these versions performs best. That version would then replace the original procedure's code. If the program continues spending a significant amount of execution time on this procedure, then the system could repeat the above process with the second level of the tree, and so forth.

In addition to DOPs and MRTEs, which perform optimization during a single invocation of a program, continuous optimization environments (COEs), which can gather statistics and reoptimize throughout an application's deployment lifetime, have been proposed [28]. A COE would actually be the ideal environment for OSE. In addition to applying OSE dynamically as described above, the COE could leverage repeated runs of the application over an extended time period in order to further explore the full configuration space, thus overcoming any suboptimal choices made during static configuration selection. One way to do this would be to obtain random points of the configuration space, as seen in Section 9.5, and use the best-performing ones as new seeds for the expansion-selection sequence of Section 8.2. The results of this process could be communicated to the original OSE compiler, in order to enhance its performance on non-COE applications and to provide a better starting point for future COE runs.

Chapter 9

OSE Experimental Evaluation

In order to evaluate the effectiveness of the OSE approach, the Electron compiler was retrofitted to implement OSE. The resulting compiler is called OSE-Electron. As mentioned earlier, Electron is the SPEC reference compiler for the Itanium platform, thus providing a credible experimental baseline.

Details of the experimental setup have already been presented in Section 7.1.2. Section 9.1 gives details on how the Electron compiler was retrofitted for OSE. Section 9.2 presents OSE-Electron's tuning process. Section 9.3 presents experimental results on OSE-Electron's compiled code performance and compile-time dilation. Section 9.4 gives more insight into the previous section's performance results by analyzing a few notable cases. Finally, Section 9.5 evaluates OSE-Electron's configuration selection and compile-time pruning methods by comparing them to a randomized configuration selector.

9.1 OSE-Electron

This section provides the implementation details of OSE in Intel's Electron compiler for Itanium. This implementation was used to produce the experimental results presented later in this section.

9.1.1 Exploration Driver

The Electron compiler's optimization framework consists of a profiler, an inlining routine, a high-level optimizer (HLO), including traditional dataflow and loop optimizations, and an optimizing code generation (CG) phase, which includes software pipelining, predication, and scheduling. Optimization proceeds as follows:

```
E1.  Profile the code.
E2.  For each procedure:
E3.      Compile to the high-level IR.
E4.      Perform a lightweight HLO pass.
E5.  Perform inlining
E6.  For each procedure:
E7.      Perform a second, more comprehensive HLO pass.
E8.      Perform code generation (CG), including software pipelining, predication,
         and scheduling.
```

In order to build OSE-Electron, we inserted an OSE driver right after inlining (step 5 above). For each procedure the driver decides whether OSE should be applied and which configurations should be tried. Thus the compilation process of OSE-Electron is as follows:

```
O1.  Profile the code.
O2.  For each procedure:
O3.      Compile to the high-level IR.
O4.      Perform a lightweight HLO pass.
O5.  Perform inlining.
O6.  For each procedure:
O7.      If the procedure is hot:
```

O8. Perform OSE on the second HLO pass and CG.

O9. Select the procedure's best optimized version for emission.

O10. If the procedure is not hot, apply Steps E7-E8 without exploration.

Since OSE-Electron is a retrofit of an existing compiler, it is not an ideal OSE implementation; it incorporates several sub-optimal implementation choices. For example, due to certain technical difficulties the exploration omits the first HLO pass and the inlining process. Also, the exploration is limited to the configuration space described in Table 7.1. A compiler built for OSE from scratch would make many more optimization parameters available for exploration. Finally, the performance estimator's success suffers from the limited profiling data that Electron makes available, as we will see later in this section.

Although OSE-Electron makes use of the profile weights gathered by the Electron compiler, it is important to note that the OSE technique is not crucially dependent on profile data. Just like any other profile-driven compiler technique, such as inlining or software pipelining, OSE could work with statically determined profile weight estimates.

9.1.2 Performance Estimation

Two factors drove the design of the static performance estimation routine in OSE-Electron. The first was compile time. Since the estimator must be run on every version of every procedure compiled, a simple and fast estimation routine is critical for achieving reasonable compile times. For this reason, the estimator chosen performs a single pass through the code, forgoing more sophisticated analysis techniques. The second limitation resulted from limited information. The final code produced by the Electron compiler is annotated with basic block and edge execution counts calculated in an initial profiling run and then propagated through all optimization phases. Unfortunately, without path profiling information many code transformations make the block and edge profiles inaccurate. Further, more sophisticated profile information, such as branch misprediction or cache miss ratios, could be useful to the estimator, but is unavailable.

Each code segment is evaluated at compile time by taking into account a number of performance indicators. The performance estimate for each code segment is a weighted sum of all such indicators. The indicators used are.

Ideal cycle count The ideal cycle count T is a code segment's execution time assuming perfect branch prediction and cache behavior. It is computed by multiplying each basic block's schedule height with its profile weight and summing over all basic blocks.

Data cache performance To account for load latencies, a function of data cache performance, each load instruction is assumed to have an average latency of λ. Whenever the value fetched by a load instruction is accessed within the same basic block, the block's schedule height, used in the computation of T above, is computed using a distance of at least λ cycles between the load-use pair.

Another term is introduced to favor code segments executing fewer dynamic load instructions. The number of load instructions executed according to the profile, L, provides another bias toward better data cache performance.

Instruction cache performance The most obvious predictor of instruction cache performance is, of course, a segment's code size C. Another performance indicator seeks to bias the estimator against loop bodies that do not fit into Itanium's first-level instruction cache. This is achieved by the formula:

$$I = \sum_{L \in \text{ loops of } S} \left\lfloor \frac{\text{size}(L)}{\text{size}(\text{L1 Icache})} \right\rfloor \times wt(L)$$

where S is the code segment under consideration and $wt(X)$ is the profile weight of X. The floor operator is used to model the bimodal behavior of loops that just fit in the cache against those that are just a bit too large.

Branch misprediction The Electron compiler does not provide us with detailed branch behavior profile information. Therefore, OSE-Electron has to approximate branch misprediction ratios using edge profiles. For each code segment S, the estimator assesses a branch misprediction penalty term according to the formula:

$$B = \sum_{b \,\in\, \text{branches of } S} \min(p_{\text{taken}}, 1 - p_{\text{taken}}) \times wt(b)$$

where p_{taken} is the probability that the branch b is taken, as determined by the edge profiles, and $wt(b)$ is the profile weight of b.

Putting it all together Given a source-code procedure F, let S_c be the version of F's code generated by a compiler configuration C, and let S_0 be the version of F's code generated by Electron's default configuration. Then the static estimation value for the code segment S_c is computed according to the formula:

$$E_c = \alpha \times \frac{T_c}{T_0} + \beta \times \frac{C_c}{C_0} + \gamma \times \frac{I_c}{I_0} + \delta \times \frac{L_c}{L_0} + \epsilon \times \frac{B_c}{B_0}$$

where terms subscripted with C refer to the code segment S_c, and terms subscripted with 0 refer to the code segment S_0. Whenever two or more versions of a code segment are compared, the one with the lowest estimation value prevails.

A brute-force grid searching method was used to assign values in the interval $[0, 1)$ to the weights $\alpha, \beta, \gamma, \delta$, and ϵ. The same search determined the load latency parameter λ. The grid search used the same sample of procedures that will be used in Section 9.2. The grid search determined the values of α, β, γ, δ, ϵ, and λ that guide the performance estimator to the best possible choices on the sample. The resulting values are: $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.001$, $\delta = 0.03$, $\epsilon = 0.0004$, and $\lambda = 2.6$.

One might assume that the design of an OSE static performance estimator for IA-64 is facilitated by the processor's in-order nature, and that it would be difficult to design similar estimators for out-of-order processors. This, however, is not the case, because of the fact that the OSE performance estimator only needs to make *relative* predictions. Take for example the load-use distance parameter λ above. Although the exact number of stalled cycles because of a cache miss is more difficult to predict on an out-of-order processor, it is still the case that a version of a code segment with greater load-use distances is *less* likely to incur stalls, and thus is preferable. Of course, the exact value of λ would have to be different. However, since parameter values are determined automatically, this would not present a problem to the compiler designer.

9.1.3 Hot Code Selection

To limit compile time, OSE-Electron limits the exploration to the proverbial 10% of the code that consumes 90% of the runtime. For this purpose, the smallest possible set of procedures accounting for at least 90% of a benchmark's runtime is determined. OSE-Electron then applies an OSE compilation process on procedures in this set, and a traditional compilation process on the remaining procedures. We experimentally verified that this fraction yields a good trade-off between compile time and performance by trying a number of other thresholds.

9.2 OSE Tuning

As described in Chapter 8, an OSE compiler needs to undergo a tuning phase, in which the configuration space is statically pruned, the configuration tree is formed, and the performance estimator is tuned. From the benchmarks described in Section 7.1.2, we chose to use the SPEC2000 suite as OSE-Electron's tuning set. More precisely, we formed a set of code samples comprising all procedures in SPEC2000 benchmarks that consume 5% or more of their benchmark's runtime. There are 63 such procedures in the SPEC2000 suite. The 5% threshold was chosen because timing measurements of procedures with too short runtimes tend to exhibit high levels of noise, which might in turn lead OSE-Electron's tuning phase to wrong choices. Procedure runtimes were obtained by running the SPEC2000 executables, using the instrumentation described in Section 7.1.2, with the training inputs specified by the SPEC2000 suite. The choice of benchmarks for the tuning set was motivated by the fact that commercial compilers are usually tuned using the SPEC2000 benchmark suite. The rest of the benchmarks mentioned in Section 7.1.2, which were omitted from the tuning set, will be used later for a fairer evaluation of OSE-Electron's performance.

The parameters described in Table 7.1 form a space of 2^{17} configurations. From these we selected 25 configurations using the methodology described in Section 8.2. We used Electron's `O2` and `O3` configurations as seeds, and we performed two iterations of the expansion and selection steps. A third iteration was aborted, because its expansion step did not produce any significant performance improvements. These 25 configurations were organized according to the methodology described in Section 8.3 into the 2-level, 3-way tree shown in Figure 9.1, which contains 12 configurations in all. Finally, the performance estimator described in Section 9.1.2 was tuned using the 63 SPEC2000 procedures in our code sample.

The progress of the tuning phase can be seen in Figure 9.2. The runtime performance of each benchmark when optimized using Electron's default configuration forms the graph's baseline. The first bar in the graph represents the performance of OSE-Electron at the end of the static selection phase, without static performance estimation or compile-time pruning. Here each procedure in a benchmark is optimized using the 25 configurations produced by the static selection phase, and the best version is selected for emission after measuring actual runtimes. The second bar represents OSE-Electron's performance employing static performance estimation, but no compile-time pruning. For the third bar, both the static estimator and the configuration tree were used. Runtimes of both procedures (in the first bar) and benchmarks were determined by the instrumentation system described in Section 7.1.3, using the benchmarks' training inputs. Using the same set of inputs for both tuning and performance measurement allows us to focus on the performance impact of OSE-Electron's features, which might be obscured by input set differences. A fairer evaluation of OSE-Electron, using separate training and evaluation inputs, will be provided in Section 9.3.

As we can see from the graph, OSE-Electron produces a 5.3% overall improvement on the performance of SPEC2000 benchmarks over Electron, IPF's SPEC reference compiler. Gains are especially pronounced for `164.gzip`, `179.art`, and `256.bzip2`. The graph also shows that static performance estimation sacrifices a modest amount of performance. This is inevitable, since static performance predictions cannot always be accurate. Interestingly, in some cases the estimator makes better choices than the actual runtime measurements. This is a result of interactions between procedures not taken into account in either experiment, but contributing to the final runtimes. While this adds a factor of uncertainty, note that the average performance improvement due to OSE is well above this factor. These runtime dependences between procedures also explain why OSE-Electron with compile-time tuning occasionally outperforms an exhaustive search of the selected configurations.

Figure 9.2 also shows that the addition of compile-time pruning sacrifices almost no performance. On the other hand, dynamic pruning causes a very significant reduction in OSE-Electron's compile time, as can be seen in Figure 9.3. This figure compares the compile times of OSE-Electron with and without compile-time pruning. The baseline for this graph is the compile time spent by Electron's default configuration. As we can see, OSE can be applied at a compile-time cost of 88.4% compared to a traditional compiler. For comparison purposes, Electron's default optimizing configuration (`-O2`) is about 200% slower than nonoptimizing compilation (`-O0`). Therefore, OSE makes iterative compilation practical enough for the general-purpose domain.

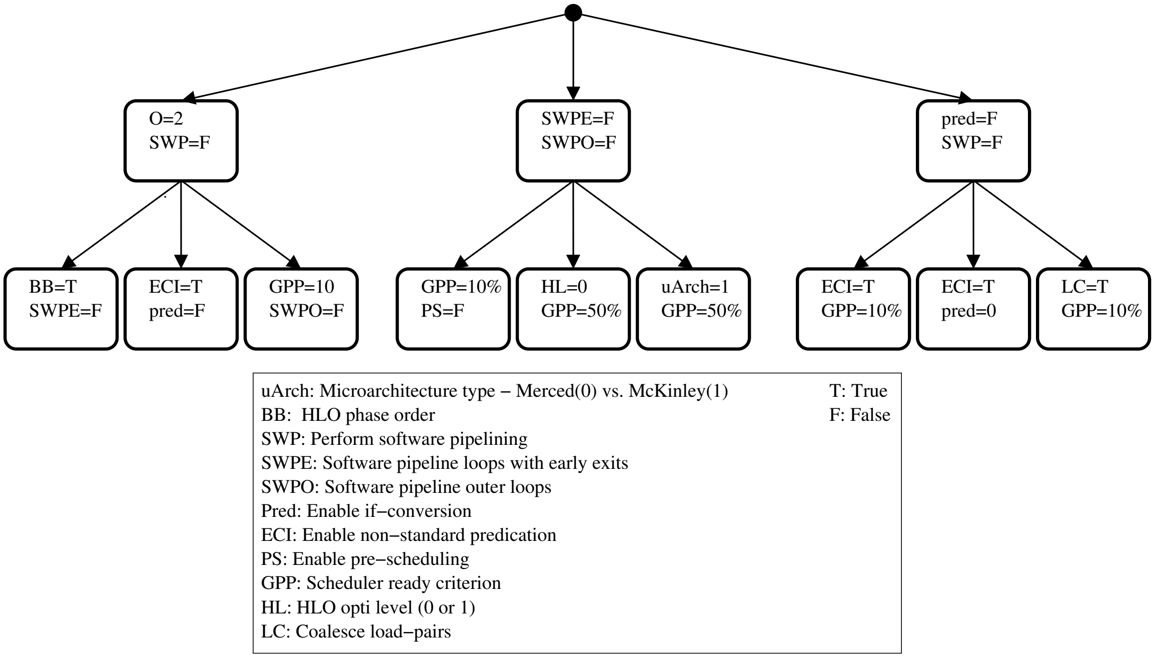

O=2
SWP=F
SWPE=F
SWPO=F
pred=F
SWP=F
BB=T
SWPE=F
ECI=T
pred=F
GPP=10
SWPO=F
GPP=10%
PS=F
HL=0
GPP=50%
uArch=1
GPP=50%
ECI=T
GPP=10%
ECI=T
pred=0
LC=T
GPP=10%
uArch: Microarchitecture type − Merced(0) vs. McKinley(1)
T: True
BB: HLO phase order
F: False
SWP: Perform software pipelining
SWPE: Software pipeline loops with early exits
SWPO: Software pipeline outer loops
Pred: Enable if−conversion
ECI: Enable non−standard predication
PS: Enable pre−scheduling
GPP: Scheduler ready criterion
HL: HLO opti level (0 or 1)
LC: Coalesce load−pairs

Figure 9.1: Tree of configurations for OSE-Electron's compile-time search.

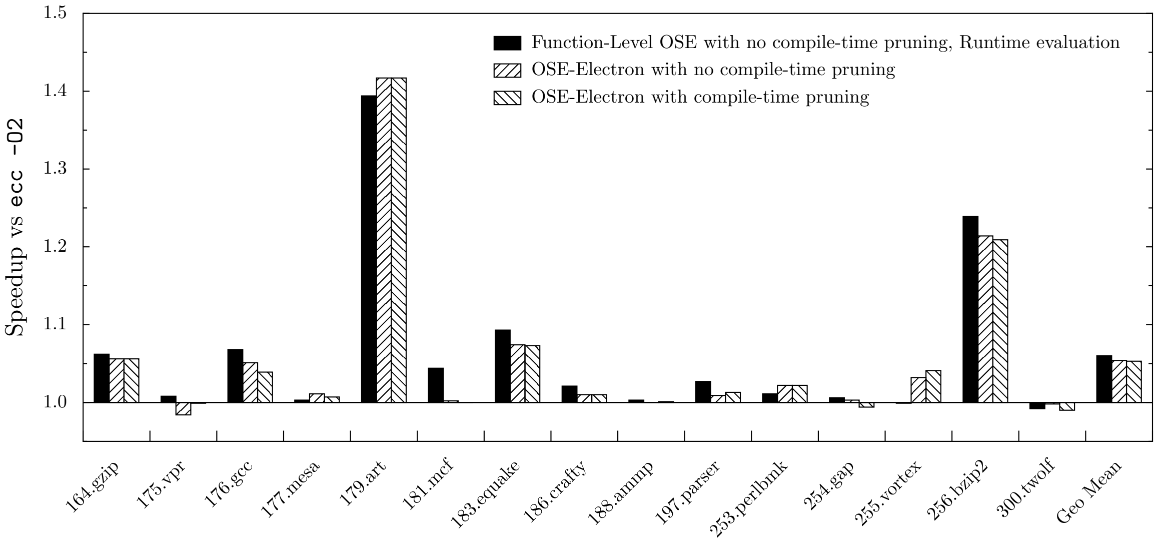


Figure 9.2: Performance of OSE-Electron generated code for SPEC benchmarks, with and without static performance estimation and compile-time pruning.

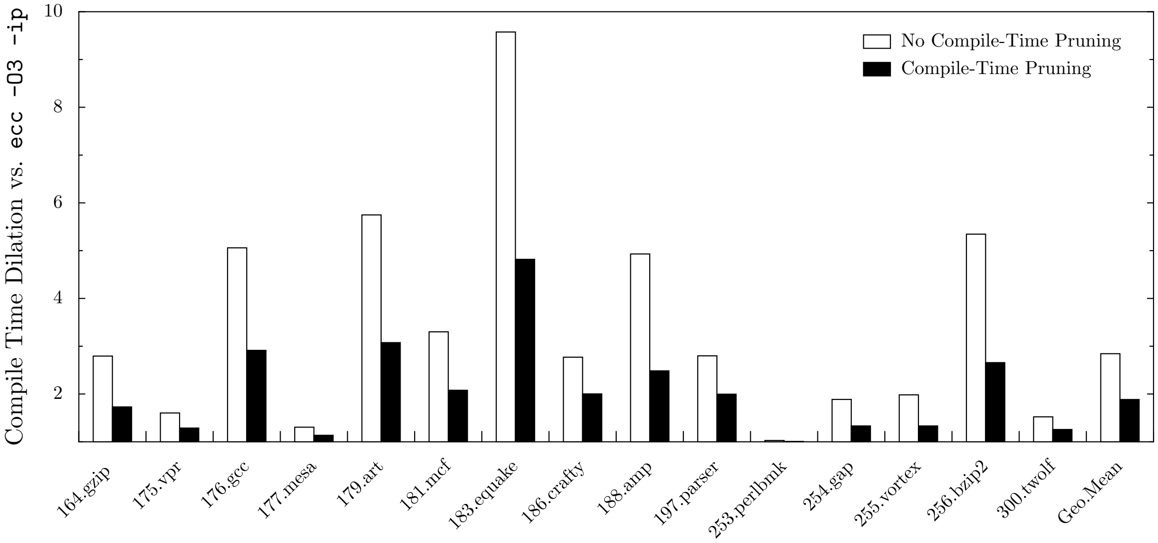


Figure 9.3: Compile time dilation for OSE-Electron over standard Electron.

9.3 OSE Performance Evaluation

To obtain a more thorough evaluation of OSE-Electron's performance benefits, we applied it on a set of benchmarks different from its tuning set. For this purpose we used the SPEC95 and MediaBench benchmarks of Section 7.1.2, as well as `yacc`. The performance improvement caused by OSE-Electron compared to Electron's default optimizing configuration can be seen in Figure 9.4. Unlike Figure 9.2, we used different training and evaluation inputs for each benchmark in this experiment. As we can see, OSE-Electron performs 10% better overall, and up to 56% better in individual cases, than Electron's default configuration.

Counter-intuitively, OSE-Electron performs better on these benchmarks than on the benchmarks in its tuning set. This can be explained by the fact that Electron's heuristics were probably tuned very carefully with the SPEC2000 suite in mind, whereas they were not as well tailored to the benchmarks tried here. OSE-Electron, on the other hand, can fit the optimization needs of both benchmark sets.

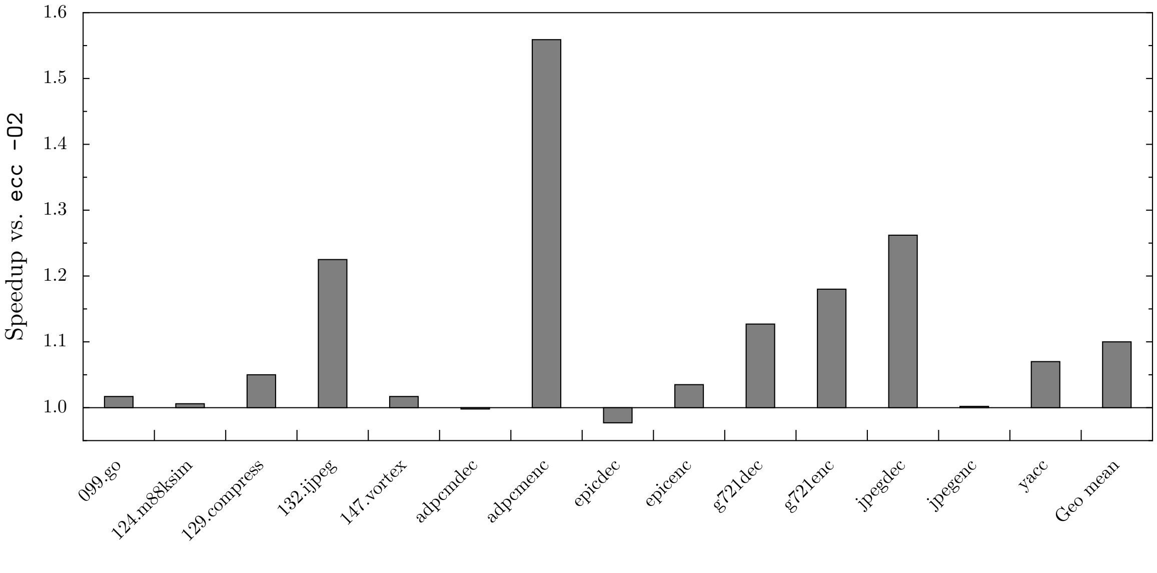


Figure 9.4: Performance of OSE-Electron generated code for non-SPEC benchmarks.

9.4 Postmortem Code Analysis

The significant performance benefits produced by OSE in many of the benchmarks tried above motivate us to look for the sources of these benefits. Below we examine three of the most prominent examples of OSE's performance improvements, and identify how the configuration exploration and the performance estimator arrived at these results.

9.4.1 SPECint95 benchmark `132.ijpeg`

Consider the procedures `jpeg_fdct_islow` and `jpeg_idct_islow` in the `132.ijpeg` SPEC95 benchmark. These procedures compute forward and inverse discrete-cosine transforms on image blocks. When compiled using Electron's default configuration, these two procedures account for about 36% of the benchmark's execution time. Each of these two procedures contains two fixed-count loops iterating 64 times.

Electron's high-level optimizer, which is run before the more machine-specific low-level optimizer in its back end, contains a loop unrolling transformation for fixed count loops, controlled by a heuristic. Since the code of the four loops described above contains many data dependencies, which would prevent efficient scheduling, the loop unrolling heuristic decides to unroll each of these loops 8 times. Subsequently, a second loop unrolling transformation in the back-end optimizer unrolls each loop another 8 times.

While full unrolling seems sensible in this case, if the high-level unrolling is turned off, `jpeg_fdct_islow` sees a 120% performance improvement, with similar results for `jpeg_idct_islow`. This is because complete unrolling makes each procedure's code bigger than the 16K level-1 instruction cache. The result is that `132.ijpeg` spends 19% of its execution time in instruction-cache stalls when the code in these procedures is fully unrolled, and only 5% when unrolling is not applied on them. This instruction cache performance loss overwhelms any gains due to better scheduling. One is tempted to think that better high-level loop unrolling heuristics could avoid this problem. However, this is unlikely, since such heuristics would have to anticipate the usually significant code size effect of all future optimization passes. On the other hand, the OSE performance estimator has the advantage of examining both loop-unrolled and non-loop-unrolled versions of the code at the end of the optimization process, where the problem with loop unrolling is easy to spot.

9.4.2 SPECint 2000 Benchmark `256.bzip2`

Another case where OSE is able to achieve a large performance benefit is the procedure `fullGtU` in the `256.bzip2` SPEC2000 benchmark. When compiled with Electron's default configuration, this procedure accounts for 48% of total running time. When software pipelining is disabled, this procedure's performance improves by 76%.

Software pipelining is applied in order to overlap iterations in a loop while yielding fewer instructions and higher resource utilization than unrolling. During software pipelining, the loop's 8 side exits are converted to predicated code. The conditions for these side exits, and consequently the conditions on the new predicate define operations in the pipelined loop, depend on values loaded from memory within the same iteration of the loop. Since the remainder of the code in the loop is now data-dependent upon these new predicates, the predicate defines end up on the critical path. To reduce schedule height, these predicate defining instructions are scheduled closer to the loads upon which they depend. During execution, cache misses stall the loop immediately at these predicate defines, causing performance degradation.

The performance of this code depends heavily on the ability of the compiler to separate these ill-behaved loads from their uses. However, the constraints governing this separation are difficult to anticipate until after optimization. In this case, the predication causing the problem only occurs after the software pipelining decision has been made. Anticipating and avoiding this problem with a predictive heuristic would be extremely difficult. On the other hand, the OSE compile-time performance estimator can easily identify the problem, since it can examine the load-use distance after optimization.

9.4.3 MediaBench Benchmark `adpcmenc`

Examining the adpcmenc benchmark reveals that over 95% of the execution time is spent in one procedure, `adpcm_coder`. This procedure consists of a single loop with a variety of control flow statements. With `-O3` turned on Electron aggressively predicates the loop yielding a 12% decrease in schedule height versus `-O2`, which leaves much of the control flow intact. This accounts for all the speedup observed. The OSE-Electron estimator can easily pick the shorter version of the code since other characteristics considered are similar between the versions. While this fact could lead one to conclude that the `O3` level is simply better than `O2`, changing Electron's default configuration to `O3` would actually lead to performance degradation for more than half the benchmarks in our suite. On the other hand, OSE-Electron is able to deliver the benefits of the `O3` configuration while avoiding its performance pitfalls.

9.5 Evaluating Configuration Selection Decisions

By following the OSE static selection and compile-time pruning methodologies, OSE-Electron is able to deliver significant performance benefits by trying just 6 configurations per code segment, 3 for each tree level. To evaluate the effectiveness of both these methodologies, we compare the OSE-Electron described above against a randomized version of OSE. This version constructs 6 configurations by assigning for each parameter in Table 7.1 a randomly picked value from the parameter's value set. Each benchmark is then compiled using these random configurations, and the best version of each procedure is selected by using OSE-Electron's static estimator. Figure 9.5 compares the performance of OSE-Electron with that of its "Monte-Carlo" version.

From the figure we can see that a randomly selected configuration set generally offers less performance benefits than the configuration set picked by OSE-Electron's selection phases. On average, the random configuration set performs about 1% worse than OSE-Electron on SPEC2000 benchmarks, and about 6% worse on the other benchmarks.

Notice that the random configuration set provides big speedups (over 15%) in only 3 benchmarks, whereas the normal OSE-Electron achieves large speedups in 7 benchmarks. The few benchmarks, particularly `256.bzip2`, where a random configuration selection performs better than one would expect occur because the performance improvements in these benchmarks are caused by varying a single optimization parameter: other optimization parameters have very little effect. In these cases each random configuration has a 25% - 50% chance of finding the correct configuration in each random trial. In the experiment above, we try 6 random configurations, meaning that it will find the correct answer with a probability between $1 - (.5)^6$ and $1 - (.25)^6$.

A similar analysis also explains the relatively modest improvement of OSE-Electron versus random configurations on the SPEC benchmark suite. Since Electron was tuned for these benchmarks, many heuristic-controlled configurations do quite well, greatly improving the random configurations' chances of generating good results. Notice that for non-

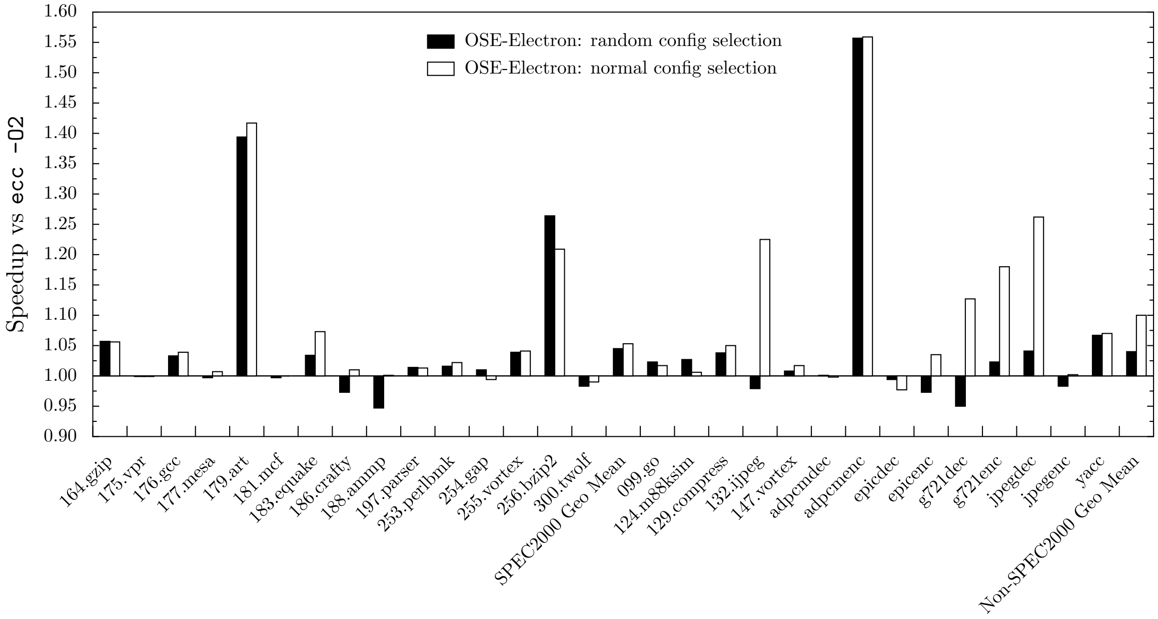


Figure 9.5: Comparison of OSE-Electron's static and compile-time configuration selection vs. random configuration selection.

SPEC benchmarks, OSE-Electron significantly outperforms the random configurations. In short, OSE is even more effective when the compiler encounters codes for which it was not tuned.

Chapter 10

Conclusion and Future Directions

This dissertation focused on two fundamental limitations that a conventional compilation framework imposes on modern optimization routines, and addressed them by proposing novel compilation techniques. Section 10.1 draws conclusions on these limitations and their proposed solutions. The dissertation ends with a discussion of future directions in Section 10.2.

10.1 Conclusions

To deal with the limited optimization scope caused by procedure-based compilation, this dissertation presented Procedure Boundary Elimination, a compilation approach that allows unrestricted interprocedural optimization. Unlike inlining, which can only extend the scope of optimization by duplicating procedures, PBE allows optimization scope and code specialization decisions to be made independently, thus increasing their effectiveness. Unlike traditional interprocedural optimization, which is constrained by having to maintain a program's procedural structure and is too costly for extensive use, PBE allows optimization to freely operate across procedures by permanently removing procedure boundaries, and allows the compiler implementor to balance performance benefits and compile-time costs through region-based compilation.

A working PBE prototype has been implemented in VELOCITY, the Liberty Group's experimental optimizing compiler. Experiments using this prototype show that PBE can achieve the performance benefits of aggressive inlining with less than half the latter's code growth and without prohibitive compile-time costs. As part of PBE's development, this dissertation also made the following individual contributions:

- An extended interprocedural analysis algorithm, necessary for processing flowgraphs generated by PBE.

- Novel region selection and region encapsulation schemes.

- A novel code duplication method, appropriate for recovering the benefits of aggressive inlining within the PBE framework.

Additionally, this dissertation experimentally demonstrated that predictive heuristics in traditional, single-path, "one size fits all" compilation approaches sacrifice significant optimization opportunities, thus motivating iterative compilation. It then proposed a novel iterative compilation approach, called Optimization-Space Exploration, that is the first such approach to be both general and practical enough for modern aggressively optimizing compilers targeting general-purpose architectures. Unlike previous iterative compilation approaches, OSE does not incur prohibitive compile-time costs. This is achieved by leveraging existing predictive heuristics, by carefully selecting the search space during compiler tuning, by utilizing feedback in order to further prune the search space at compile time, and by relying on a fast static performance estimator for generated code evaluation.

The potential of OSE has been experimentally demonstrated by implementing an OSE-enabled version of Intel's aggressively optimizing production compiler for Itanium. Experimental results from this prototype confirm that OSE is capable of delivering significant performance benefits while keeping compile times reasonable.

10.2 Future Directions

Although the experimental evaluations in Chapters 6 and 9 have demonstrated the value of both PBE and OSE, significant opportunities for improvement exist for both methods.

PBE could benefit from more sophisticated region formation methods, especially methods that concentrate on dataflow properties and “optimizability” measures rather than profile weights. Additionally, PBE’s targeted code specialization phase could be enhanced with more code duplication techniques, in order to more effectively control code growth and/or increase performance.

As for OSE, a further exploration of how OSE can fit within managed runtime environments and dynamic compilation systems (Section 8.5) would be promising. Also, OSE’s tuning and runtime exploration phases could benefit by incorporating ideas from the use of artificial intelligence compilation approaches, such as the one described in [48].

More importantly, PBE and OSE are currently being applied on single-threaded codes only. However, both techniques have significant potential in the increasingly important domain of coarse-grained parallelism extraction. Preliminary experiments with thread-level parallelism techniques, such as the one proposed by Ottoni et al. [42], show that these techniques could significantly benefit by an increase in optimization scope, such as the one offered by PBE. Since these techniques mostly focus on loops, PBE’s ability to handle traditional loops and recursion in a uniform way can also offer significant benefits. However, several challenges have to be overcome before TLP optimizations can be incorporated in the PBE framework. The most important practical problem is that region formation heuristics have to produce parallelism-friendly regions. This is very different from producing regions that are suitable for more traditional optimizations. Another problem is that, although larger regions offer more opportunities for parallelism, they may make certain aspects of the TLP extraction algorithm, especially load balancing, prohibitively expensive. To overcome this limitation, the idea of using multiple levels of regions is being explored. Instead of having to consider the entire CFG of a region, the TLP extraction algorithm could then examine a “reduced” CFG of sub-regions, and break up or descend into individual sub-regions only when necessary. These ideas are being actively pursued by other members of the Liberty Research Group.

Applying OSE to the TLP domain is more straightforward, since good TLP heuristics are particularly hard to craft. Probably the biggest challenge would be to modify OSE’s performance estimator so that it can take TLP performance parameters, such as load balancing and thread synchronization points, into account.

Bibliography

[1] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoggerbrugge, P. Hu, W. Jalby, P. M. W. Knijnenburg, M. F. P. O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. Stohr, M. Verhoeven, and H. A. G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proceedings of the European Conference on Parallel Processing*, pages 1351–1356, 1997.

[2] An evolutionary analysis of GNU C optimizations, December 2003. http://www.coyotegulch.com/acovea/index.html.

[3] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the 1988 Conference on Programming Language Design and Implementation*, pages 241–249, Atlanta, GA, June 1988.

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

[5] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 92–103, 1997.

[6] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.

[7] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computation. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[8] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, 1991.

[9] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the 1986 Symposium on Compiler Construction*, pages 152–161, July 1986.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–370, May 1992.

[11] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 1999 Symposium on Principles of Programming Languages,* pages 133–146, January 1999.

[12] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 2000 Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[13] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[14] A. Eichenberger, W. Meleis, and S. Maradani. An integrated approach to accelerate data and predicate computations in hyperblocks. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 101–111, November 2000.

[15] S. Eranian. Perfmon: Linux performance monitoring for IA-64. http://www.hpl.hp.com/research/linux/perfmon/, 2003.

[16] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing*, College Park, MD, July 2002.

[17] J. R. Goodman and W. C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, July 1988.

[18] J. Goubault. Generalized boxings, congruences and partial inlining. In *First International Static Analysis Symposium*, Namur, Belgium, September 1994.

[19] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[20] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[21] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

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

[23] W. A. Havanki. Treegion scheduling for VLIW processors. Master's thesis, Department of Computer Science, North Carolina State University, 1997.

[24] W. W. Hwu and P. P. Chang. Inline function expansion for compiling realistic C programs. In *Proceedings of the 1989 Conference on Programming Language Design and Implementation*, pages 246–257, June 1989.

[25] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[26] Intel Corporation. *Electron C Compiler User's Guide for Linux*, 2001.

[27] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[28] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2001.

[29] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In *Proceedings of the 1999 International Symposium on High Performance Computing*, pages 121–132, 1999.

[30] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction*, pages 125–140, Paderborn, Germany, October 1992.

[31] J. Knoop and B. Steffen. Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. Technical Report 9309, Institut fur Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet zu Kiel, April 1993.

[32] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the 2004 Conference on Programming Language Design and Implementation*, pages 171–182, New York, NY, USA, 2004. ACM Press.

[33] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, October 2000.

[34] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, 1999.

[35] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez. Modulo scheduling with reduced register pressure. *IEEE Transactions on Computers*, 47(6):625–638, 1998.

[36] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[37] E. Morel and C. Renvoise. Interprocedural elimination of partial redundancies. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 160–188. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[38] W. G. Morris. CCG: A prototype coagulating code generator. In *Proceedings of the 1991 Conference on Programming Language Design and Implementation*, pages 45–58, Toronto, ON, CA, June 1991.

[39] E. W. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th Symposium on Principles of Programming Languages*, pages 219–230, Jan. 1981.

[40] A. P. Nisbet. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.

[41] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.

[42] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th International Symposium on Microarchitecture*, November 2005.

[43] Perfmon: An IA-64 performance analysis tool, April 2002. http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4.

[44] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 49–61, June 1995.

[45] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proceedings of the 1995 Colloquium on Formal Approaches in Software Engineering*, pages 651–665, May 1995.

[46] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

[47] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[48] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *Proceedings of the European Conference on Genetic Programming*, Essex, UK, April 2003.

[49] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the 2003 Conference on Programming Language Design and Implementation*, pages 312–323, June 2003.

[50] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006.

[51] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *Journal of Instruction-Level Parallelism*, 7, 2005.

[52] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the '03 International Symposium on Code Generation and Optimization*, pages 204–215, March 2003.

[53] T. Way, B. Breech, and L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 24, May 2000.

[54] T. Way and L. Pollock. A region-based partial inlining algorithm for an ILP optimizing compiler. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 705–710, Cambridge, MA, 2002.

[55] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19:1053–1084, November 1997.

[56] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 274–286, December 1996.

[57] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the 2003 Conference on Languages, Compilers and Tools for Embedded Systems*, pages 1–11, New York, NY, USA, 2003. ACM Press.