A Framework for Unrestricted Whole-Program Optimization

Spyridon Triantafyllis Matthew J. Bridges Easwaran Raman Guilherme Ottoni David I. August

Department of Computer Science
Princeton University
{strianta,mbridges,eraman,ottoni,august}@princeton.edu

Abstract

Procedures have long been the basic units of compilation in conventional optimization frameworks. However, procedures are typically formed to serve software engineering rather than optimization goals, arbitrarily constraining code transformations. Techniques, such as aggressive inlining and interprocedural optimization, have been developed to alleviate this problem, but, due to code growth and compile time issues, these can be applied only sparingly.

This paper introduces the Procedure Boundary Elimination (PBE) compilation framework, which allows unrestricted whole-program optimization. PBE allows all intra-procedural optimizations and analyses to operate on arbitrary subgraphs of the program, regardless of the original procedure boundaries and without resorting to inlining. In order to control compilation time, PBE also introduces novel extensions of *region formation* and *encapsulation*. PBE enables *targeted code specialization*, which recovers the specialization benefits of inlining while keeping code growth in check. This paper shows that PBE attains better performance than inlining with half the code growth.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization; D.3.3 [*Programming Languages*]: Language Constructs and Features—Procedures, functions, and subroutines

General Terms Experimentation, Performance, Theory, Algorithms

Keywords whole-program analysis, whole-program optimization, interprocedural analysis, interprocedural optimization, procedure unification, region-based compilation, region formation, region encapsulation, specialization, superblock, inlining, path-sensitive analysis

1. Introduction

Compiler support is essential for good performance on modern architectures. In addition to the traditional tasks of simplifying computations and eliminating redundancies, an aggressively optimizing compiler must efficiently exploit complex computational resources, expose instruction-level parallelism (ILP) and/or thread-level parallelism (TLP), and avoid performance pitfalls such as memory stalls and branch misprediction penalties. To meet these challenges, a compiler relies on a rich set of aggressive optimization and analysis routines. However, traditional procedure-based analysis and optimization approaches greatly hamper the ability of these routines to produce efficient code because the original breakup of a program into procedures serves software engineering rather than optimization goals.

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. This 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 [13, 17, 19] 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. 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 very frequently executed call sites with very small callees. Section 2 will examine the benefits and shortcomings of inlining and interprocedural optimization in greater detail.

This paper introduces *Procedure Boundary Elimination* (PBE), a compilation framework that allows unrestricted whole-program optimization, to overcome the limitations of procedure-based compilation. 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 [19], 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. The WCFG building phase is described in Section 3.1.

To avoid the costs of optimizing the whole program as a single overly-large unit, PBE applies *region formation* [9, 20, 21]. 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

PLDI '06 June 10–16, 2006, Ottawa, Canada

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 Section 3.2.

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. A detailed discussion of PBE's code specialization routines can be found in Section 3.4.

After procedure unification, region formation, and targeted code specialization, subsequent optimization and analysis phases are presented with compilation units that are very different from procedures. PBE regions are multiple-entry multiple-exit, can have arbitrary dataflow across their boundaries, and may contain call and return arcs that have a many-to-many correspondence, as illustrated in Section 4. Effective optimization requires that such compilation units be accurately analyzed. The PBE analysis framework achieves this by extending the interprocedural analysis algorithms presented by Sharir and Pnueli [19].

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 Section 5 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 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.

2. Related Work

A significant body of prior work in the fields of interprocedural optimization and analysis, full and partial inlining, and region formation is related to PBE because of shared goals and methodologies. This section describes this prior work and highlights how PBE differs from or expands upon it. A detailed discussion of related work on code specialization methods can be found in Section 3.4.

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 [13, 15, 17, 19]. A path on an interprocedural control-flow graph is considered realizable if its call and return transitions are in accordance with procedure call semantics [19]. 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. The interprocedural dataflow analysis problem is exponential in the general case [15, 19]. 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 [4, 17]. As we will see in Section 4, the PBE analysis algorithm expands on the works cited above.

Using the results of interprocedural analysis, some classical optimization routines can be performed interprocedurally (for example, see [4, 18]). Obviously, an interprocedural optimization routine is able to exploit many more optimization opportunities than its intraprocedural counterpart. 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. Additionally, extending a variable's live range across a call or return can be tricky, as data exchange between procedures is normally possible only through the parameter-passing mechanism or through memory-resident global variables. 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 (Section 3) and addresses the problem of excessive compile-time costs through region-based compilation (Section 4).

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 research compilers in order to expose additional optimization opportunities [2, 5, 11]. 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. Although inlining provides significant performance benefits, it also causes 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, inlining experiments cited in Chang et al. [5] show a 11% overall performance improvement at the cost of 17% code growth. One undesirable side effect of excessive code growth is that optimization and analysis time may increase significantly. Hank et al. [9] report a more than eightfold compile-time increase when 20% of a benchmark's call sites are inlined. To avoid these pitfalls, compilers usually apply inlining only to very frequent call sites with very small callees, thus limiting the technique's applicability and value.

Partial inlining [8, 20, 21] alleviates traditional inlining's code growth problems by duplicating only a portion of the callee into the call site. This is achieved by removing infrequently executed parts of the callee and repackaging them as separate procedures. Implementations of partial inlining in just-in-time compilers 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 [20]. 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. This makes these transitions much costlier, which in turn makes partial inlining worthwhile only for procedures containing sizeable parts of very infrequently executed code. This restriction is even

more pronounced in [20], where a transition from hot to cold code forces recompilation.

The drawbacks of inlining, either full or partial, essentially stem from its procedure-based nature. Full inlining is constrained by the fact that it can only duplicate whole procedures. Partial inlining is more flexible, but it is also constrained by the fact that it must result in a program neatly divided into procedures. This is true even for region-based partial inlining techniques, such as [20, 21]. Although these techniques form regions interprocedurally, they still must copy and rearrange the code so that each region is entirely contained in a single procedure when the partial inlining phase concludes. This leads to the restrictions outlined above. By removing procedure boundaries and allowing the compiler to operate on arbitrary portions of the program, PBE removes these restrictions.

2.3 Region-Based Compilation

Region formation [9] was originally proposed to cope with the excessive compile-time dilation that occurs when optimizing the large procedure bodies produced by aggressive inlining. A region is essentially a compiler-selected, multiple-entry multiple-exit portion of the 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 itself to operate within a single region at a time. Experiments by Hank et al. [9] show that region-based compilation can achieve radically reduced compile times at the cost of only minimal performance loss. Subsequent research, such as [20, 21] discussed above, incorporates region formation with full or partial inlining for greater effect.

PBE is also a region-based compilation technique. The crucial difference with prior region-based techniques is that PBE-produced regions can be arbitrary program segments, potentially spanning parts of multiple procedures. In comparison, previous techniques either form regions only within existing procedures [9], or form regions across procedures but then apply inlining so as to eliminate calls and returns within regions [20, 21], thus suffering from the limitations discussed in Section 2.2. As we will see in Section 3.2, the approach taken by PBE increases the compiler's flexibility and effectiveness, but also presents new challenges to the rest of the optimization process.

3. Procedure Boundary Elimination

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. The overall flow of the compilation process in PBE can be seen in Figure 1.

A PBE compiler begins by applying the following three phases, explained in detail in Sections 3.1 to 3.4.

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.

Targeted Code Specialization (TCS) This phase is applied separately within each region. It identifies sites in the region where code specialization is likely to provide optimization opportunities and duplicates code accordingly.

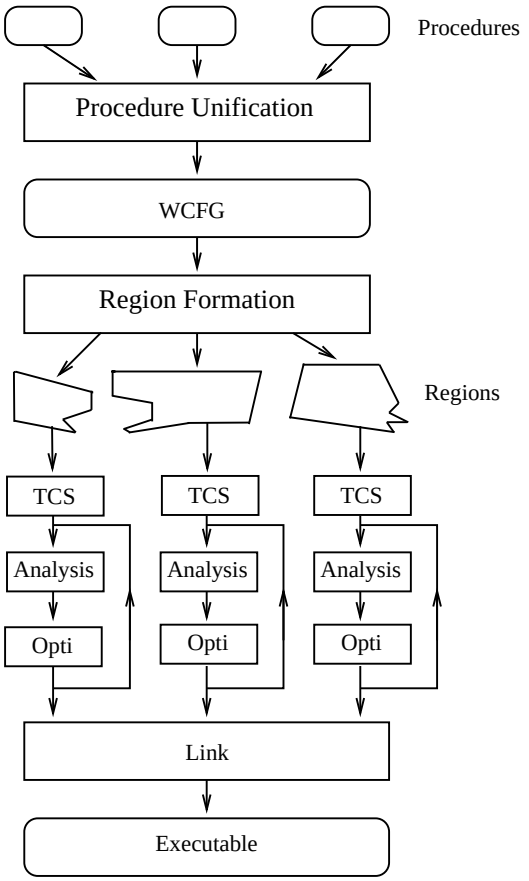


Figure 1. Overview of PBE compilation flow.

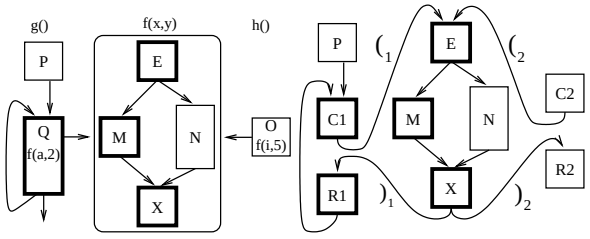
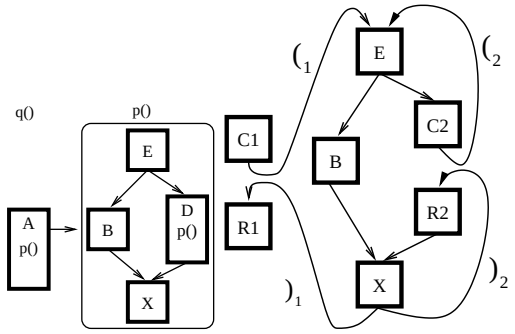


Figure 2. A code example (a) before and (b) after unification.

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, covered in Section 4.

3.1 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 program, 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



(a) Before unification (b) After unification

Figure 3. Recursive procedure (a) before and (b) after unification.

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 [19], 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 calls 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 2. Figure 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 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 Section 4.

Perhaps more interesting is the example in Figure 3, which shows the result of applying procedure unification to a recursive procedure. After unification (Figure 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 action dictated by the calling convention. Interprocedural optimizations respect these semantics and work around them appropriately, although this complicates or even hinders their application. Respecting the calling convention is necessary, since these routines must preserve a program's division into procedures. 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, 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 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 moves of actual parameter values into 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 moves.
- 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 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.

3.2 Region Formation

After unification, the rest of the code transformations described in this paper could operate on the whole program. Indeed, this would enable these 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 program size.

Region formation solves this problem by breaking up the program 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 regions will be superior compilation units to the program's original procedures, which are chosen according to criteria unrelated to optimization. Indeed, previous research [9] indicates that performance loss due to region formation is minimal.

The remainder of this section describes the profile-based region formation heuristic used by our initial implementation of PBE, presented in Section 5. This heuristic is very different from the one originally proposed by Hank et al. [9]. 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 an inter-region transition 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 inter-region 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 CFG 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 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 phase to the heuristic. In this phase, any cluster whose size is less than a certain percentage 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 Section 5, 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. More details about the performance of the region formation heuristic presented above can be found in Section 5.

3.3 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. [9], 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

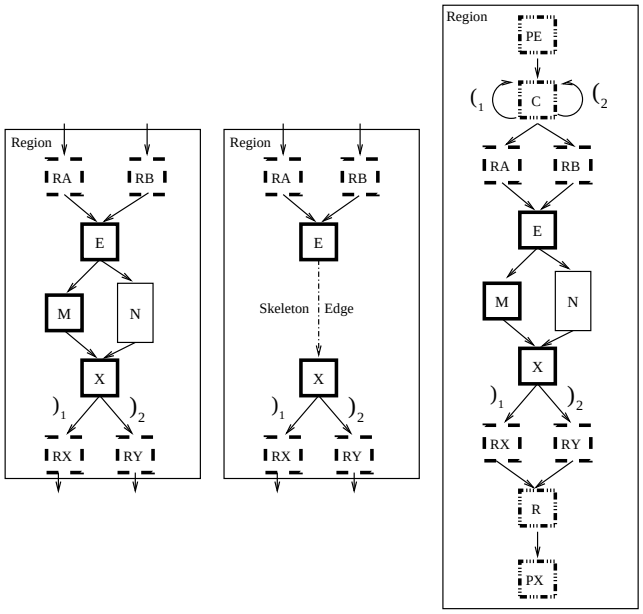


Figure 4. (a) A region formed around blocks from Figure 3b with nodes for region entry and exit. (b) its program skeleton form, with blocks M and N abstracted into a skeleton edge. (c) the encapsulated form of the region.

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 Section 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 abbreviate 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.

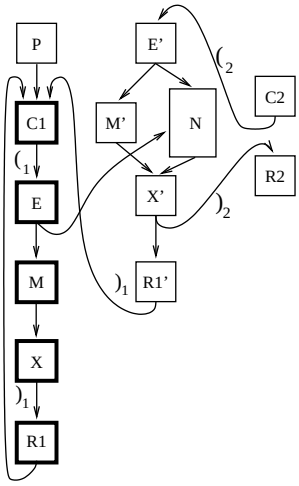


Figure 5. The code of Figure 2b after superblock formation.

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 Section 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. Figure 4 shows a region that has been conveted into its program skeleton form. Blocks M and N are converted into a single skeleton edge that summarizes their effects.

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 added to represent the program entry. This node is followed by a node C that has edges of the form $C \xrightarrow{(_i} C$ circling back to it, for every return annotation $)_i$ that appears in the region[1]. 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. Region exits are handled in a symmetric way. Figure 4c shows a region after it has been encapsulated. Since return annotations $)_1$ and $)_2$ are unmatched at the region entries, edges containing call annotations $(_1$ and $(_2$ are added to C.

With regions thus encapsulated, further analysis and optimization routines do not need to consider the entire WCFG, avoiding scalability problems in further compilation phases.

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 [3, 6, 10, 12, 14], 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.

Superblock formation

Superblock formation [12] 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 2 can be seen in Figure 5. 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 Section 5, 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.

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 Section 5 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:

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

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 se-

[1] In fact, PE 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.

ries 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 Section 5, $\alpha = 0.1$. 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 Section 5, $\beta = 50\%$.

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 Section 5 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.

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 (especially [10] and [3]) can also be applied in the PBE compiler. Actually, any such method is bound to increase its impact in PBE, since PBE enables it to work across procedures.

4. Dataflow 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 2b 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.

For this reason, we developed a *context-sensitive* analysis approach for PBE. The PBE analysis algorithm presented in this section is derived from the *functional approach* to interprocedural analysis [19]. An analysis method following this approach works by calculating *transfer functions* between each procedure entry and each CFG node in the procedure. When the transfer function between a procedure entry and its exit is discovered, it can be annotated on *summary edges* linking each call site to that procedure with the corresponding return site. Transfer functions on summary edges essentially show how an analysis value is transformed due to a call. Therefore, the analysis value at a return site can be determined by applying the transfer function of the corresponding summary edge to the analysis value of the corresponding call site. The transfer function of a procedure depends on the transfer function of its callees, therefore all such transfer functions have to be calculated through simultaneous iteration. After this part of the calculation is complete, the interprocedural analysis routine uses the transfer functions between each procedure entry and each call node in the procedure to determine the analysis values of all procedure entries. After the analysis values of all procedure entries are known, the analysis values of all nodes can be trivially calculated by applying the transfer functions from the corresponding procedure entry to the node to the analysis value of the procedure entry. See [19] and [17] for more information.

PBE analysis faces several challenges that are not present in a classical interprocedural analysis and therefore are not handled by the above algorithm. 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 5, both return arcs $X \overset{)_1}{\rightarrow} R_1$ and $X' \overset{)_1}{\rightarrow} R'_1$ match the single call arc $C_1 \overset{(_1}{\rightarrow} E$. This situation never appears in classical interprocedural analysis, where call and return arcs always have a one-to-one correspondence. (This is the case in Figure 2b, where optimizations have not yet been applied.) 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 CFG into procedure-like constructs for the purposes of analysis will also be a task for the analysis algorithm. The algorithm presented in the rest of this section meets these challenges, while staying as close as possible to the traditional functional interprocedural analysis algorithm.

In the following presentation, we will assume for simplicity that we have a forward analysis problem, such as dominators or reaching definitions. Backward analysis problems, such as liveness or upwards exposed uses, can be treated in a symmetric way. In the following discussion, the symbol $\sqcup$ will be used to denote the analysis problem's confluence operator. The same symbol will be used for the confluence operator on the induced lattice of transfer functions, defined simply as follows:

$$(F_1 \sqcup F_2)(x) = F_1(x) \sqcup F_2(x)$$

The symbol $\top$ is used to denote the maximum element of the analysis problem's lattice (normally used to signify an "uninitialized" analysis value). The corresponding maximum element on the lattice of transfer functions will be $F_\top$, defined as follows:

$$F_\top(x) = \top$$

Below we present the PBE analysis algorithm as a series of 8 steps. Of these, steps 1 to 5 have to do with the structure of the CFG, and thus are independent of the specific analysis problem. Therefore these steps only need to be calculated whenever the CFG changes. Otherwise they can be reused for multiple analysis runs.

Step 1: CFG node classification

CFG 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} (in a C program, the header of the `main()` function) will also be considered a context entry. Similarly, the program exit X_{main} (the return statement of `main()`) will be considered a context exit. Note that these definitions are not mutually exclusive. For example, the node C used in region encapsulation (Section 3.3) 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, there must exist a pair of edges $C \overset{(_i}{\rightarrow} E$ and $X \overset{)_i}{\rightarrow} R$ for some value of i. In a prepass, 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). 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: $\overset{SC}{\rightsquigarrow}$). A CDP (E, X) is called *proper* if there is a same-context path $E \overset{SC}{\rightsquigarrow} X$. For each such 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 $C \overset{S_i}{\rightarrow} R$ for every pair of edges $C \overset{(_i}{\rightarrow} E$ and $X \overset{)_i}{\rightarrow} R$.

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 by 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 X. Context membership canthen 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 $C \overset{(_i}{\rightarrow} E_2$ and a matching return edge $X_2 \overset{)_i}{\rightarrow} R$. 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 of the context graph.

Step 6: Computing same-context transfer functions

For every CDP (E, X), this phase calculates a transfer function $F_{E,N}$ from the entry E to every node $N \in CT_{E,X}$. This transfer function will summarize the effect on the analysis values of all the same-context paths linking E and N. This is accomplished by running a simple, meet-over-all-paths dataflow analysis pass on the nodes of each context separately. Since this dataflow analysis runs on same-context paths, which may contain summary edges, we will need to assign transfer functions to summary edges for this process to work. This is done as follows: Originally, every summary edge is assigned the transfer function $F_\top$. Whenever the transfer function $F_{E,X}$ between the entry and the exit of a CDP is discovered, this function is assigned to all summary edges belonging to that CDP. Since these summary edges may be parts of same-context paths in other contexts, they may affect the transfer functions of other CDPs, leading to the discovery of the transfer functions of other summary edges. This process has to be applied iteratively until it converges. This is similar to the phase in classical interprocedural analysis that calculates the transfer functions between each procedure entry and each node in the procedure.

Step 7: Computing context entry values

When Step 6 terminates, the compiler has the transfer function from each context entry to every call site in that context. The compiler can use this information to calculate transfer functions from context entries to other context entries. For example, consider two contexts CT_{E_1,X_1} and CT_{E_2,X_2} such that $CT_{E_1,X_1} \rightarrow CT_{E_2,X_2}$. Let $C_1, C_2, \ldots, C_n$ be all the call sites that lead from CT_{E_1,X_1} to CT_{E_2,X_2}. That is, for every C_i there is a path $E_1 \overset{SC}{\rightsquigarrow} C_i$ and a call edge $C_i \overset{(_i}{\rightarrow} E_2$. Then the transfer function from E_1 to E_2 is the confluence of the F_{E_1,C_i} transfer functions:

$$F_{E_1,E_2} = \bigsqcup_i F_{E_1,C_i}$$

Using these transfer functions, the compiler can calculate the analysis values at all entries. This is done by annotating each transfer function F_{E_1,E_2} on the corresponding edge $CT_{E_1,X_1} \rightarrow CT_{E_2,X_2}$ of the context graph, and then running an iterative dataflow analysis pass on the context graph. This phase is similar to the second phase in functional interprocedural analysis, which calculates the analysis values at all procedure entries.

Step 8: Computing node values

Now, the compiler has the analysis values at all context entries (from step 7) as well as all the transfer functions from context entries to same-context nodes. Therefore, calculating the analysis values at all nodes is easy. Suppose that node N belongs to contexts $CT_{E_1,X_1}, \ldots, CT_{E_n,X_n}$. Then its analysis value is:

$$v_N = \bigsqcup_i F_{E_i,N}(v_{E_i})$$

Again, this is similar to the last phase in functional interprocedural analysis.

Complexity

For locally separable problems, efficient interprocedural dataflow analysis has a complexity of $O(ED^2)$ [17], where E is the number of edges in the interprocedural CFG and D is the number of dataflow "facts" (registers in liveness analysis, definitions in reaching definitions analysis, etc.). For most analyses both E and D are $O(n)$, where n is the number of nodes in the CFG, resulting in a worst-case complexity of $O(n^3)$. In practice, the complexity of a dataflow analysis is known to be roughly quadratic on the size of CFG nodes.

As far as complexity is concerned, the crucial difference between PBE and classical interprocedural analysis is that, while in the former case each CFG node belongs to a single procedure, in the latter case each CFG node may belong to multiple contexts, thus causing certain actions to be repeated (especially in Step 6). Thus the resulting worst-case complexity of PBE would be $O(\alpha n^3)$, where α is the maximum number of contexts any node in the CFG belongs to. In the worst case, $a = O(n)$. However, multiple contexts per node arise mostly due to the code specialization routines in Section 3.4. Due to their targeted nature and their code-growth limits, these routines are likely to leave most of the code alone, and copy the rest of the code only a modest number of times. Thus a is more likely to be a value only slightly greater than 1. To recap, PBE analysis for locally separable problems has a worst-case complexity of $O(n^4)$, but is expected to behave quadratically in practice.

5. Experimental Evaluation

In order to evaluate the ideas presented in this paper, we compared the performance of a PBE-enabled compilation process to a process using aggressive inlining. For that purpose we used our own experimental compiler, called VELOCITY, combined with the IMPACT

research compiler [1]. IMPACT is the best-performing compiler for the Itanium architecture. In our experimental setup IMPACT is used as a front end, compiling C benchmarks to a low-level IR called Lcode. Benchmarks used in the experiments were taken from the SPEC CINT95 and CINT2000 benchmark suites.

Each benchmark is profiled using training inputs before being lowered. IMPACT also annotates Lcode with the results of aggressive alias analysis [16], which are exploited in later optimization phases. Lcode is then used as input to VELOCITY, which implements all further optimization and analysis routines. VELOCITY contains an aggressive classical optimizer which 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. VELOCITY also includes superblock formation, superblock-based scheduling, and register allocation. Finally, it performs inlining for one set of experiments, and unification, region formation, and area specialization for the other. Whenever possible, the heuristics of the VELOCITY compiler closely follow those of IMPACT, especially for inlining, register allocation, and scheduling.

However, 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 bundling constraints into account. Finally, in the above compilation process, both register allocation and scheduling are performed assuming an Itanium 2 target machine.

Immediately after the Lcode is input into VELOCITY, it undergoes a first pass of classical optimization. This "cleans up" the code, making subsequent inlining and TCS heuristics more effective. From then on, each benchmark's compilation follows three different paths. In the first path no inlining or procedure unification is performed. The results of this path form the baseline in our measurements. In the second path, the code is subjected to a very aggressive inlining pass, copied from the IMPACT compiler. 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 scheduling.

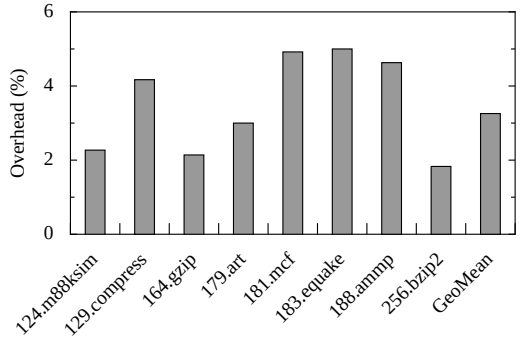


Figure 6. Overhead of unification, region formation, region encapsulation, and area specialization as a percentage of total compilation time.

Figure 6 shows the percent of total compile time spent performing unification, region formation, region encapsulation, and area specialization. This is essentially the up-front overhead of applying PBE. The full overhead of PBE, which also includes compile-time dilation in later optimization phases, will be presented in Figure 10.

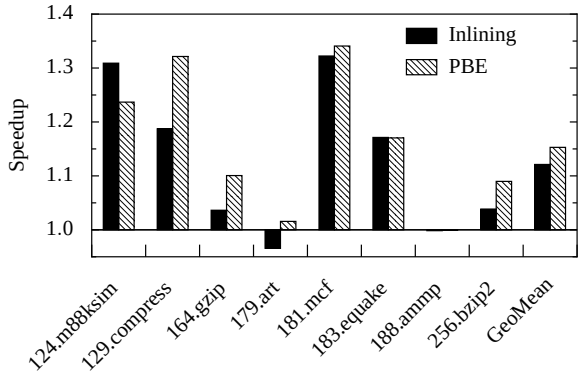


Figure 7. Performance benefits of inlining and PBE on training inputs over strict procedure-based compilation (dynamic cycle count).

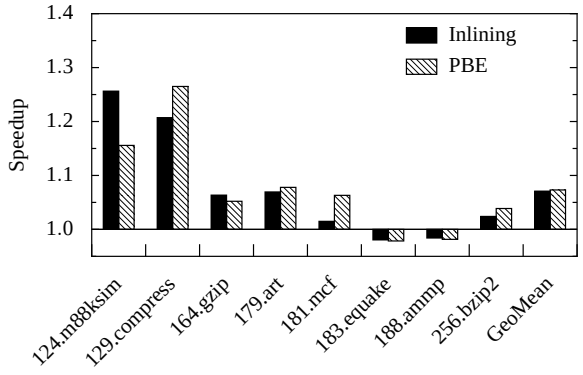


Figure 8. Performance benefits of inlining and PBE on training inputs over strict procedure-based compilation (runtime performance).

As seen in Figure 6, 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 Section 3 do not cause a large increase in the compilation time. In particular, use of the program skeleton during region encapsulation avoids the excessive overhead that occurs when analyzing the whole program.

To measure the quality of the code produced, we collected the dynamic cycle count and runtime performance numbers using training inputs. Training inputs were used to obtain an accurate assesment of the capabilities of PBE versus inlining, avoiding the issues of profile accuracy. The results were obtained by profiling or running the application after all optimizations, translation to Itanium code, and register allocation and scheduling were performed. To obtain dynamic cycle count, the schedule height of each basic block is multiplied with the block's execution count. This is equivalent to simulating the code on an Itanium-like uniform 6-wide VLIW machine with perfect cache and branch predictor behavior. The runtime performance numbers were collected using *pfmon* [7]. Executables were run on an HP workstation zx2000 with a 900Mhz Intel Itanium 2 processor and 2Gb of memory running Redhat Advanced Workstation 2.1.

The results of the performance experiments are shown in Figures 7 and 8, representing dynamic cycle count and runtime performance, respectively. Additionally, code size and compile-time were measured. Figure 9 measures the code growth caused by in-

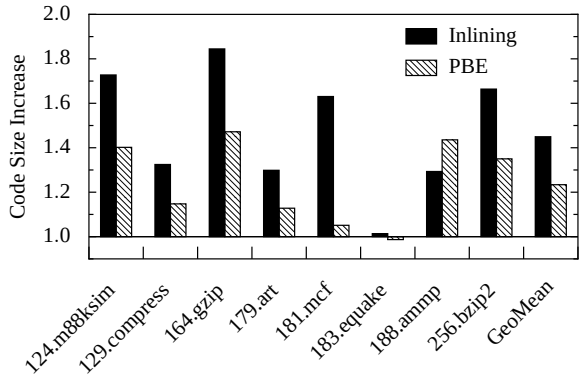


Figure 9. Code growth caused by inlining and PBE compared to strict procedure-based compilation

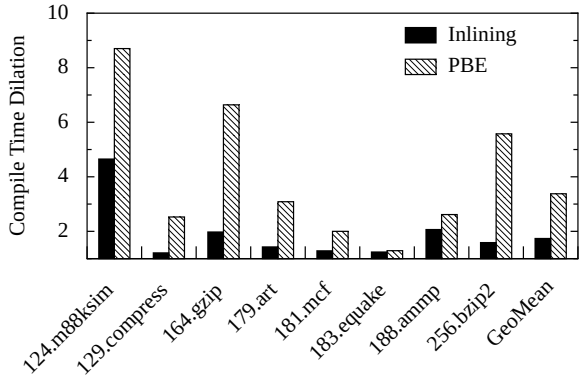


Figure 10. Compile time dilation for inlining and PBE compared to strict procedure-based compilation

lining and PBE compared to the baseline, while Figure 10 measures the compile-time cost incurred by PBE and inlining over the base compilation.

As the dynamic cycle count graph (Figure 7) shows, PBE performance beats aggressive inlining, 15% vs. 13% on average. PBE's performance advantage is much more pronounced in certain individual cases, such as `129.compress` (32% vs. 19%) and `164.gzip` (10% vs. 3.6%). Similar performance gains are shown in the runtime performance graph (Figure 8), though the overall speedup is about half the gain estimated from dynamic cycle count.

Most importantly, PBE achieves these performance benefits with only about half the code growth of inlining, 23% vs. 45% on average. 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 inlining in this experiment. 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 vs. code growth restrictions are even more pronounced in 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 region boundaries.

Figure 10 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 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 limiting most classical optimizations to within basic blocks or capping the number of optimization passes applied, 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.

6. Conclusion

In this article, we 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 preliminary experimental evaluation of PBE shows that it can achieve the performance benefits of aggressive inlining with less than half the latter's code growth and without prohibitive compile-time costs.

Apart from the PBE compilation technique itself, this paper contains the following individual contributions:

- An extended interprocedural analysis algorithm, necessary for processing PBE-generated flowgraphs (Section 4).
- Novel region selection and region encapsulation schemes (Sections 3.2 and 3.3).
- A novel code duplication method, appropriate for recovering the benefits of aggressive inlining within the PBE framework (Section 3.4).

In the future, we plan to evaluate more sophisticated region formation methods, especially methods that concentrate on dataflow properties instead of profile weights. In addition, we intend to investigate novel targeted code specialization methods to more effectively control code growth and/or increase performance. We are also investigating the applicability of PBE to thread-level parallelism extraction.

Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. Additionally, we thank Andrew Appel, Sharad Malik, David Walker, Brian Kernighan for their help with this work. Finally, we thank the anonymous reviewers for their insightful comments. This work has been supported by the National Science Foundation (NGS-0133712 and NGS-0305617) and Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation or Intel Corporation.

References

[1] AUGUST, D. I., CONNORS, D. A., MAHLKE, S. A., SIAS, J. W., CROZIER, K. M., CHENG, B., EATON, P. R., OLANIRAN, Q. B., AND HWU, W. W. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture* (June 1998), pp. 227–237.

[2] AYERS, A., SCHOOLER, R., AND GOTTLIEB, R. Aggressive inlining. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (June 1997), pp. 134–145.

[3] BODIK, R., GUPTA, R., AND SOFFA, M. L. Complete removal of redundant computation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (June 1998), pp. 1–14.

[4] CALLAHAN, D., COOPER, K. D., KENNEDY, K., AND TORCZON, L. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction* (July 1986), pp. 152–161.

[5] CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., AND HWU, W. W. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience 22*, 5 (May 1992), 349–370.

[6] EICHENBERGER, A., MELEIS, W., AND MARADANI, S. An integrated approach to accelerate data and predicate computations in hyperblocks. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (November 2000), pp. 101–111.

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

[8] GOUBAULT, J. Generalized boxings, congruences and partial inlining. In *First International Static Analysis Symposium* (Namur, Belgium, September 1994).

[9] HANK, R. E., HWU, W. W., AND RAU, B. R. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture* (December 1995), pp. 158–168.

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

[11] HWU, W. W., AND CHANG, P. P. Inline function expansion for compiling realistic C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (June 1989), pp. 246–257.

[12] HWU, W. W., MAHLKE, S. A., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A., OUELLETTE, R. G., HANK, R. E., KIYOHARA, T., HAAB, G. E., HOLM, J. G., AND LAVERY, D. M. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, 1 (January 1993), 229–248.

[13] KNOOP, J., AND STEFFEN, B. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction* (Paderborn, Germany, October 1992), pp. 125–140.

[14] MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., BRINGMANN, R. A., AND HWU, W. W. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture* (December 1992), pp. 45–54.

[15] MYERS, E. W. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM symposium on Principles of programming languages* (Jan. 1981), pp. 219–230.

[16] NYSTROM, E. M., KIM, H.-S., AND HWU, W.-M. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium* (August 2004).

[17] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (June 1995), pp. 49–61.

[18] SANTHANAM, V., AND ODNERT, D. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation* (June 1990), pp. 28–39.

[19] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 189–233.

[20] SUGANUMA, T., YASUE, T., AND NAKATANI, T. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation* (June 2003), pp. 312–323.

[21] WAY, T., BREECH, B., AND POLLOCK, L. Region formation analysis with demand-driven inlining for region-based optimization. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques* (May 2000), p. 24.