A Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler*

Tom Way and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{way,pollock}@cis.udel.edu

Abstract

Region-based optimization is an effective technique for restructuring a program to better reflect dynamic behavior and increase interprocedural optimization and scheduling opportunities in an ILP compiler. In this paper, we describe an algorithm that incorporates partial inlining into a region-based compilation framework to achieve the optimization benefits of full inlining, but with the added benefit of reduced code growth.

Keywords: inlining, region-based ILP optimization

1 Introduction

Interprocedural techniques for program analysis and optimization have become increasingly important to exploit the higher degrees of available instruction-level parallelism (ILP) in modern architectures. Region-based compilation [4] is a generalized trace selection approach for ILP that reduces the cost of aggressive interprocedural analysis and optimization by partitioning a program into units of compilation, or *regions*, based on profile information. By applying procedure inlining, where a procedure call site is replaced by the body of the called procedure, and restructuring a program into regions, the region-based compiler obtains more freedom to perform code motion and other analyses and optimizations interprocedurally, while maintaining control over the compilation unit size and content. Unlike traditional procedure-based compilation, region-based techniques provide a method for bounding the size of the unit of compilation to better control optimization costs [4].

The key component of a region-based compiler is the region formation phase which partitions the program into regions using profile-guided heuristics with the intent that the ILP optimizer will be invoked with a scope that is limited to a single region at a time. Thus, the quality of the generated code depends greatly upon the ability of the region formation phase to create regions that a global optimizer can effectively transform in isolation for improved instruction-level parallelism.

Procedure inlining plays an important role in enabling optimizations that cross procedure boundaries by making it possible for the region formation phase to easily analyze across procedure boundaries and form *interprocedural regions*, which consist of instructions from more than one procedure of the original program. Procedure cloning, where a new, uniquely named copy of a procedure is created and one or more call sites renamed to call the clone, also has been used to reduce the impact of code growth due to inlining [8].

A major shortcoming of inlining and cloning techniques is that they do not discriminate between frequently and infrequently executed portions of a procedure; the entire procedure body is inlined or cloned. Interprocedural scope is gained through inlining, but unnecessary code growth results from inlining infrequently executed code, which can prohibit the full exploitation of ILP and runtime performance improving optimizations. Although cloning can produce less code growth than inlining and can enable call site specific optimizations, it duplicates both frequently and infrequently executed code and does not increase interprocedural scope.

To address these pitfalls, this paper presents a region formation algorithm that incorporates *partial* procedure inlining to enable the compiler to inline only the frequently executed portions of a procedure. The originators of region-based compilation [4] suggested that region-based compilation creates a natural framework for partial inlining. Challenges to partial inlining are described, and an implementation of the algorithm within the Trimaran ILP research compiler [6] is evaluated.

*Supported in part by the National Science Foundation Grant EIA-9806525.

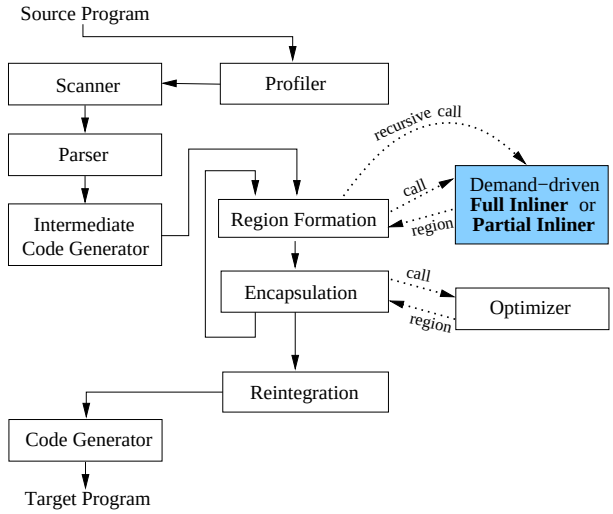


Figure 1: Organization of a region-based compiler framework with partial inlining.

2 Region-based Partial Inlining

2.1 The partial inlining problem

The **partial inlining problem** can be defined as the problem of modifying the procedure restructuring technique of full inlining to be more beneficial to compilation, optimization and, ultimately, runtime performance by reducing code growth while improving the average execution frequency of the code being inlined. The promise of partial inlining includes the potential for speeding up execution, increasing the interprocedural scope and numbers of instructions seen by the scheduler, improving optimization opportunities, controlling code growth, and improving the profile homogeneity (i.e., similar profile weights of instructions within a region). These benefits mirror those of full inlining. Partial inlining has the added potential for reducing code growth as compared to full inlining by reducing the quantity of infrequently executed code that is inlined.

Partial inlining is performed by leveraging the ability of region formation to restructure procedures into profile-related regions of code. The result of traditional region formation is to group instructions within a procedure based on profile weight; instructions tend to be placed in regions with other instructions of similar profile weight. The region with the most frequently executed portion of a procedure is the *hot* region, while the remaining regions are designated as *cold*.

Figure 1 illustrates the incorporation of demand-driven partial inlining into region-based compilation framework [7, 9]. After a traditional front end, including a profiler, scanner, parser and intermediate code generator, region formation is performed. During region formation, regions are formed within a procedure by selecting basic blocks based on profile weight. First, a seed block is selected with maximum profile weight in a procedure. Then, all successor blocks which have a profile weight of at least 50% of the seed are added to the region. Similarly, all predecessor blocks are added, followed by all successor blocks of any block already in the region. When a block is included in a region, if it contains a procedure call, region formation is performed recursively in the callee procedure. The result of this recursive region formation can be partially or fully inlined, or not inlined at all, into the currently forming region in the caller.

As each region is completed, it is encapsulated to appear to the optimizer to be a procedure, and passed to the optimizer. After this region-based optimization, region formation continues within the procedure until all regions are formed. Once region formation and optimization is complete, regions are reintegrated back into the original procedure, and compilation proceeds in the compiler back end, including code generation.

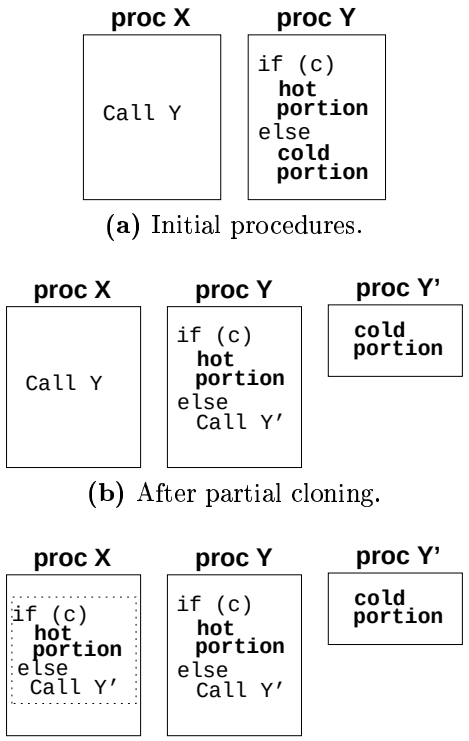
The goal of partial inlining within region formation is to remove all cold regions from a procedure, leaving behind just the hot region. Cold regions are removed by cloning, creating a new procedure for each cold region and replacing the cold region with a call site to the clone. The resulting "partial" procedure, containing the hot region and any remaining control structure and new call sites, is then eligible to be inlined.

2.2 Challenges

Issues to consider when designing a region-based partial inlining algorithm, and the techniques needed to address them, are:

- **Separating hot and cold portions** - Cloning each cold region can repartition the code based on profile information. This cloning step is called *partial cloning*. Figure 2 illustrates how partial cloning enables partial inlining. The original procedures `X` and `Y` are shown in Figure 2a. There is a call to procedure `Y` in `X`, and the more and less frequently executed portions of `Y` are denoted as hot and cold portions, respectively.

  Partial cloning is shown in Figure 2b, with the clone of the cold portion in `Y` being procedure `Y'`. The code in `Y` that was partial cloned is replaced with a procedure call to the clone `Y'`. When there are multiple single-entry, single-exit cold portions of code in the procedure, each of these cold portions is cloned to create a separate,



(a) Initial procedures.

(b) After partial cloning.

(c) After partial inlining.

Figure 2: Partial inlining through partial cloning.

uniquely named procedure and replaced by a call to the appropriate clone in the original procedure containing the cold portion. Note that the hot portion of the code remains in its original place in the control structure of `Y`, and is not cloned.

- **Inlining the hot region only** - Once the cold portions of a procedure are cloned, the remaining restructured procedure, consisting of the hot portion, its control structure and calls to the clones of the cold portions, is inlined. This inlining step is simply full inlining of this restructured procedure; it is called *partial* inlining because only part of the original `Y` is inlined into its caller.

  Figure 2c illustrates the inlining step of partial inlining. In this final step, the restructured procedure `Y` is inlined into `X` at its call site. The result of this partial inlining is that the interprocedural scope improves over the original procedure structure because of inlining in general, profile homogeneity improves because the code being inlined is of a high execution frequency, and code growth is less than full inlining due to inlining less code.

- **Handling variables and parameters** - When each partial clone is created, its new procedure definition must include parameters corresponding to all of the local variables and formal parameters of the original procedure that are being used in the original procedure outside the cloned portion and also used at any point in the code being cloned. Since changes to any of the clone's parameters made inside the clone should have an effect outside the clone, they must be passed by reference rather than by value. Any local variables or formal parameters of the original procedure that are passed into the partial clone as parameters are redefined as references (pointers), if necessary, and any uses of their values within the cloned code are redefined as (pointer) dereferences.

- **Selecting partial versus full inlining** - The additional cost and complexity of partial inlining at compile time must be balanced with the simplicity, but extra code growth, of full inlining. Region size and profile information can be used to strike this balance.

  When the sizes and profile weights of the hot and cold portions are similar, and the benefits of performing partial inlining versus full inlining are less clear, the decision about which technique to use becomes a more difficult one to make. Heuristics to determine whether partial or full inlining will be more beneficial should simultaneously consider code growth and optimization opportunities gained. Future directions for research include exploring sophisticated partial inlining heuristics.

- **Consistency of partial inlining** - Because profile information is generalized per procedure, partial inlining will be performed identically at each call site. Call site specific partial inlining can be advantageous when the runtime behavior of the callee is different at different call sites; path spectra profiling [8] is one technique that accomplishes this.

- **Maintaining original, full version of procedure** - The cloning step of partial inlining restructures the original procedure while maintaining its semantics. As a result, there in no requirement to maintain a copy of the original, full version of each procedure. The exceptional situation would be if different partial inlining of the procedure was to be performed for different call sites. With call site specific partial inlining behavior, the original version should be maintained as it would provide the full inlining option and enable different partitionings for different partial inlining options.

3 The Partial Inlining Algorithm

Figure 3 presents an algorithm for partial inlining within region formation. The algorithm shows how each call site is handled during region formation performed in a procedure `X`. At each call site to some procedure `Y`, the determination is made to perform partial, full, or no inlining. Among the factors considered are size of the procedure, profile weight, size of the hot region compared to the overall procedure, and more general inlining heuristics such as presence of recursion, a mismatch of the number or type of procedure arguments, and overall program code growth.

When it is determined that partial inlining should be performed, region formation identifies the hot region and cold regions in procedure `Y`. Each cold region is cloned with its code in `Y` replaced by a procedure call to the new clone. The transformed `Y`, containing primarily the hot region, is fully inlined into `X`. Effectively, `Y` is partially inlined due to the transformation.

In the case where the decision is made to perform full inlining, inlining of `Y` into `X` is performed on demand, and region formation continues in the larger procedure `X`. When no inlining is performed, region formation continues in the unmodified `X`.

Once region formation is complete in procedure `X`, each region is encapsulated as a procedure (to appear to the optimizer as a whole procedure), and is passed to the optimizer as a unit. The optimized region is then reintegrated back into the original procedure, `X`, and compilation continues with code generation.

```
At each call site in X to procedure Y:
if Y is partial inlinable
    form hot region
    form remaining cold regions
    foreach cold region
        create new clone procedure
        replace code with call site
    inline remaining procedure, including hot region
else if Y is fully inlinable
    inline Y into X
    continue forming regions in new X (including Y)
else
    do not inline Y
    form remaining regions in X
foreach formed region in X:
    encapsulate the region as a procedure
    optimize the encapsulated procedure
    reintegrate region into X
    emit code
```

Figure 3: Partial inlining algorithm.

The primary benefit of performing partial inlining is that the transformed procedure, in this case `Y`, can be significantly smaller than the original. Thus, the impact on code growth is significantly reduced. Additionally, because the transformed `Y` consists of the hot region, it is primarily frequently executed code, resulting in improved optimization opportunities in the portion of the procedure that should benefit the most. Optimization of the cold regions that were cloned from `Y` can be avoided if desired, due to lower profile weight, saving additional compilation and optimization time.

4 Implementation and Evaluation

The partial inlining algorithm was implemented within the Trimaran research compiler framework for ILP, which was previously modified to perform demand-driven full inlining [9]. The impact of the partial inlining algorithm on region formation was measured by compiling six benchmarks (*008.espresso*, *023.eqntott*, *124.m88ksim*, *130.li*, and *132.ijpeg* from SPEC and *clinpack* from Netlib). A number of compile-time and runtime metrics were measured to compare the full and partial inlining strategies for demand-driven inlining within region formation.

Partial inlining led to a decrease of 3% to 15% in average procedure size, due to the effect of partial cloning of cold regions, as compared with the use of full inlining within region formation. Code growth was similarly affected, with partial inlining leading to from 1% to 6% less growth in overall intermediate code size, with the exception of one smaller benchmark (*clinpack*) which had increased code growth by 1% over the full inlining strategy. Overall, the partial cloning step of partial inlining increased the number of procedures from 4% to 33%, while producing insignificant increases in the original program size of, at most, 5%, and typically 1% or less. The side effect of partial cloning was a universal drop in the interprocedural scope of each procedure, since cloning removes code from, rather than inserts code into, the procedure.

Runtime performance was improved by partial inlining over full inlining by 0.06% to 0.98%, although in one case performance worsened by 0.10% (*124.m88ksim*) due to fewer partial clones being formed as compared to the large procedure size, which led to less beneficial inlining being performed.

Partial cloning also led to a decrease in the number of regions formed, with an increase in the average size of those regions. This increased size is reflected by an increase in profile homogeneity, with more profile-

related code grouped together by profile weight than when full inlining was performed. The slight increase in profile homogeneity supports the slight increase generally seen in runtime performance.

5 Related Work

Some research has addressed partial inlining as a component of other compilation and optimization techniques [1–3, 5]. Partial inlining is included as part of an analysis technique for detecting and eliminating redundant memory allocations and associated deallocations inside a common module of a functional language program [3]. Closely related is a framework, designed for functional programming languages and based on lambda calculus, that uses a form of procedure splitting that resembles partial inlining to separate the hot and cold portions of a procedure [5].

The Dynamo dynamic optimizing compiler [1] makes use of reusable *code fragments* that it identifies at runtime as a way to eliminate redundant optimization. These code fragments are cached in optimized form and later linked into the code at runtime. In effect, this is partial inlining performed at runtime.

Procedure abstraction is used in a code compression technique for compiling programs for reducing the size of the compiled code for embedded processors while increasing optimization opportunities [2]. Profiling information and pattern-matching techniques are used at compile time to identify and coalesce repeated sequences of instructions. Rather than inlining these repeated, or *hot*, instruction sequences, a single copy is formed as a procedure through *procedure abstraction*. The original instruction sequences are each replaced with a call to the abstracted procedure. While procedure abstraction effectively is the inverse of inlining in that it removes, rather than inserts, copies of the code, it succeeds at the same goal of reducing code growth by isolating the hot code region while improving optimization.

6 Conclusions

The results of this work indicate that use of partial inlining within a region-based compilation framework potentially can improve the runtime performance of a program as compared to the use of full inlining prior to optimization. Nearly all measures of desirable improvements to program characteristics and runtime behavior were improved by partial inlining. In the experiments, the heuristics used by the demand-driven inliner during interprocedural region formation did indeed inline some of the cold regions after all hot regions had been inlined, which increased code growth but did not appear to adversely affect the performance of this modest-sized benchmark. The ability to suppress inlining of cold regions is more important in larger programs where the inlining of many cold regions could reduce the gains in code locality that result from partial inlining. Any loss of code locality could negatively impact the ability of the instruction scheduler to increase ILP, leading to reduced performance. A more comprehensive study of larger programs is planned.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, Canada, June 2000.

[2] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[3] J. Goubault. Generalized boxings, congruences and partial inlining. In *1st Static Analysis Symposium (SAS '94)*, number 864 in Lecture Notes in Computer Science, pages 147–161. Springer Verlag, Namur, Belgium, Sept. 1994.

[4] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: Introduction, motivation, and initial experience. *International Journal of Parallel Programming*, 25(2):113–146, Apr. 1997.

[5] S. Monnier and Z. Shao. Inlining as staged computation. Technical Report TR-1193, Department of Computer Science, Yale University, Mar. 2000.

[6] A. Nene, S. Talla, B. Goldberg, and R. M. Rabbah. Trimaran - an infrastructure for compiler research in instruction-level parallelism - user manual, 1998. `http://www.trimaran.org`. New York University.

[7] T. Way, B. Breech, W. Du, and L. Pollock. Demand-driven inlining heuristics in a region-based optimizing compiler for ILP architectures. In *International Conference on Parallel and Distributed Computing and Systems*, pages 90–95, Anaheim, California, 2001.

[8] T. Way, B. Breech, W. Du, V. Stoyanov, and L. Pollock. Using path-spectra-based cloning in region-based optimization for instruction-level parallelism. In *14th International Conference on Parallel and Distributed Computing Systems*, pages 83–90, Richardson, Texas, 2001.

[9] T. Way, B. Breech, and L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 24–33, Philadelphia, Pennsylvania, Oct. 2000.