Evaluation of 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, USA
email: {way,pollock}@cis.udel.edu

ABSTRACT

An ILP optimizing compiler using a region-based approach restructures a program to better reflect dynamic behavior and increase interprocedural optimization and scheduling opportunities. Regions provide the compiler with better control of the unit of compilation than traditional procedure-based compilers. In this paper, we evaluate an algorithm that incorporates partial inlining into a region-based compilation framework to achieve the optimization benefits of full inlining, with the added benefit of reduced code growth. Results are presented for a variety of common and novel metrics developed for comparison of regions, including profile homogeneity, percentage of invariant code and interprocedural scope of regions.

KEYWORDS

Parallel and distributed compilers, algorithm design, region-based ILP compiler optimization, partial inlining

1. Introduction & Background

Advanced instruction-level parallel (ILP) computer architectures require interprocedural techniques for program analysis and optimization to exploit available parallelism. An approach for ILP that reduces the cost of aggressive interprocedural analysis and optimization is region-based compilation [4]. Region-based compilation is a generalized trace selection approach that partitions a program into units of compilation, or *regions*, based on profile information. Using 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 in a region-based compiler to reduce the impact of code growth due to inlining [8].

Since traditional inlining and cloning techniques do not discriminate based on execution frequency, 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 other 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.

The originators of region-based compilation [4] suggested that region-based compilation creates a natural framework for *partial* inlining. With region-based partial inlining, the compiler attempts to inline only the frequently executed portions, or regions, of a procedure. The potential benefits of partial inlining include runtime performance improvements, increased interprocedural scope including the scope seen by the instruction scheduler, improved control of code growth, and improved profile homogeneity (i.e., similar profile weights of instructions within a region). The benefits of partial inlining mirror those of full inlining, with the added potential for further reducing code growth.

Figure 1 illustrates the incorporation of demand-driven partial inlining into a region-based compilation framework [7], which improves upon the traditional method that relied on an aggressive inlining phase in which all inlining was performed prior to region formation [4]. After a traditional compiler front end, including a profiler, region formation is performed.

Regions are formed within a procedure during re-

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

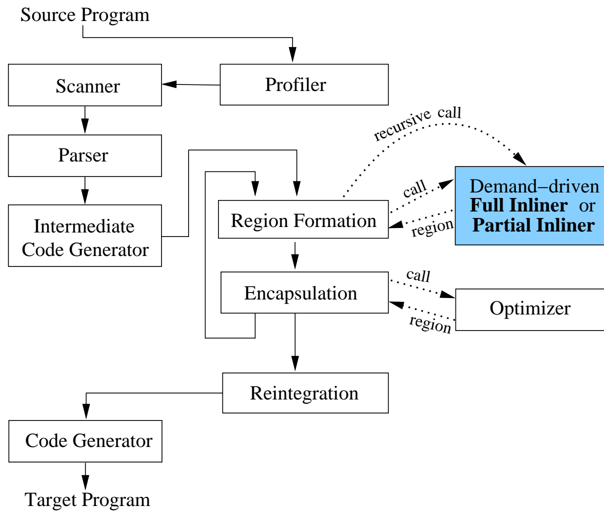
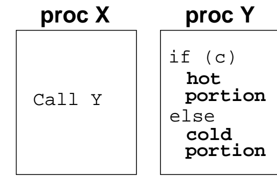
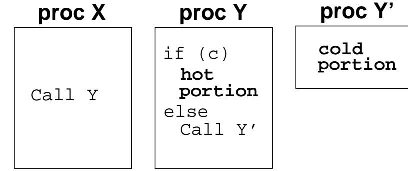


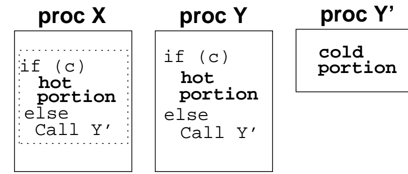
Figure 1. Organization of a region-based compiler framework with partial inlining.



(a) Initial procedures.



(b) After partial cloning.



(c) After partial inlining.

Figure 2. Partial inlining through partial cloning.

gion formation by selecting basic blocks based on profile weight. First, a seed block of maximum profile weight is selected. 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. If a block that contains a procedure call is included in a region, region formation is performed recursively, on demand, in the callee procedure. The results 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 as a procedure and passed to the optimizer. Region formation continues after this region-based optimization until all regions are formed, at which point the regions are reintegrated into the original procedure. Compilation then proceeds in the compiler back end, including the instruction scheduling and code generation phases.

A major goal of partial inlining within region formation is to remove all infrequently executed, or *cold*, regions from a procedure, leaving behind just the frequently executed, or *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.

In this paper, the design of a region formation algorithm that incorporates partial procedure inlining is described, and an implementation of the algorithm within the Trimaran ILP research compiler [6] is evaluated. Novel metrics are introduced which compare the characteristics of regions.

2. Partial Inlining Algorithm

Figure 2 illustrates the partial inlining algorithm. 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 is used to identify hot and cold regions (Figure 2a). Each cold region is cloned (Figure 2b), a technique called *partial cloning*. Partial cloning is the inverse of inlining; code is removed from procedure `Y` to form its own new procedure `Y'`, and is replaced by a call to this new clone. Effectively, `Y` is partially inlined due to the transformation (Figure 2c).

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.

The result of this partial inlining is that the interprocedural scope improves over the original procedure struc-

ture 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. As with the general forms of cloning and inlining, formal parameters, arguments and local variables may need to be renamed and/or redirected (i.e., pass-by-reference) to maintain correctness.

3. Empirical Evaluation

The partial inlining algorithm was implemented within the Trimaran ILP research compiler framework, 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). Three compilations were performed for each benchmark: (1) the original, non-cloned version of the benchmark, without performing any inlining, (2) the original, non-cloned version, using demand-driven inlining, and (3) the partial cloned version, using demand-driven inlining.

3.1 Metrics

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. The metrics used measured characteristics that were dynamic, related to compile-time behavior and runtime performance, and static, reflecting aspects of the unit of compilation. Results are presented either as directly-measured values, or as comparative values expressed by the general equation for percentage change for the new (region-based) compilation as compared with results of an original (procedure-based) compilation, as:

$$pct\ change = \frac{new_val - original_val}{original_val} \times 100 \qquad (1)$$

Performance measures

Performance of the algorithm was measured in terms of the code growth produced at compile time, and of the execution time and processor utilization of the compiled program at runtime.

The increase in the size of code that results from full or partial inlining, or **code growth**, was measured as the percentage change over a procedure-based compilation strategy that applied no inlining. Partial cloning introduces a fixed amount of code growth per clone, as the cloned code is encapsulated as a new procedure. Full and partial inlining lead to an increase in code size proportional to the cumulative size of the code that is inlined.

The runtime performance improvements seen with full or partial inlining were measured in terms of the percentage change in **execution time** of the two methods versus a procedure-based compilation.

Processor utilization is the measure of how well the compiler is able to schedule instructions for the available processor resources. The average number of instructions executed per execution cycle was measured, and analyzed in terms of the percentage change as compared to the procedure-based compilation.

Compilation unit characteristics

In procedure-based compilation, the unit of compilation is the procedure, or function. Region-based compilation seeks to restructure procedures, reorganizing them into more homogeneous and equal-sized units of compilation, or regions. The impact of region-based optimization is reflected in changes to the unit size, profile homogeneity, amount of profile invariant code, and interprocedural scope of the unit of compilation.

One of the main goals of region-based compilation is to control the size of the unit of compilation. When the **compilation unit size** is controlled, the time spent by the compiler on optimization of each unit can be controlled as well. Controlling optimization time leads to an improvement in the scalability of compilation. Units of compilation are measured by counting the number of intermediate-level instructions in a program after it has been compiled. The size of the average unit of compilation gives a value that can be used to estimate how much time the compiler will spend to optimize each unit, since many optimization phases can take quadratic time in the size of the unit being analyzed and optimized (i.e., $O(n^2)$).

Profile homogeneity is defined as the measure of how similar (i.e., using standard deviation) the given unit of compilation, either procedure or region, is in terms of profile weight per instruction. This measure of variance indicates how region formation may impact optimization. More profile homogeneous compilation units enable the optimizer to easily identify and isolate heavily executed regions, and then selectively focus more attention on these more important regions and less attention elsewhere. This partitioning reduces the chance of leaving important portions of the code unoptimized or spending excessive time optimizing unimportant code. The average profile homogeneity indicates how consistent the profile weights are within each of the benchmarks' units of compilation, calculated as the mean standard deviation of the profile weights for all compilation units in the benchmark.

The **percentage of invariant units** in the benchmark provides an indication of what proportion of the compilation units are *significantly invariant* (i.e., standard deviation of 0.01 or less). This low degree of variance means that the basic blocks within a compilation unit are profile homogeneous.

When specifically comparing regions to procedures, and regions formed using different techniques, the **interprocedural scope** is a useful measure of the change in the

Table 1. Percentage change in code growth, execution time and processor utilization for the full and partial inlining over a procedure-based strategy.

	Code growth		Exec. time		Proc. util.	
Benchmark	full	partial	full	partial	full	partial
008.espresso	17	15	-3.90	-4.21	-2.11	-2.05
023.eqntott	18	15	-6.11	-6.92	3.77	3.68
124.m88ksim	15	13	-9.22	-9.11	0.00	0.13
130.li	6	5	-12.53	-12.84	-4.70	-4.58
132.ijpeg	17	11	-10.24	-11.22	4.01	4.31
clinpack	15	16	-5.35	-5.41	-3.63	-3.79
average	15	12	-7.89	-8.28	-0.44	-0.38

amount of interprocedural instructions within the compilation units. Interprocedural scope is the ratio of interprocedural instructions to all instructions in a program.

3.2 Results and discussion

The percentages by which the code size and the average procedure size changed as a result of the restructuring performed in the initial partial cloning step were minimal. The increase in size was 5% or less in all cases, and typically 1% or less. Partial cloning led to an increase in source code size by the amount of additional code necessary to create the partial clones. This additional code includes the new clone procedure definition and call site. In general, about 5 lines of source code were added for each partial clone created, with an average of 20 clones per program, ranging relative to program size from 3 to 46. Along with this very small increase in code size came a more significant decrease in the average procedure size, ranging from 3% to 15%, due to the increase in the numbers of procedures introduced by cloning. Although compilation time could not be accurately measured due to limitations in the implementation of the research framework, compilation seemed to be faster when partial inlining was used, perhaps due to reduced code growth.

Code growth

The first column of Table 1 shows the percentage change in code growth for the benchmarks comparing region formation using demand-driven full inlining, and the use of partial inlining in addition to full inlining, calculated using Equation 1. With the exception of *clinpack*, the inclusion of partial inlining controlled code growth better than using only full inlining; code growth was reduced by from 1% to 6%. The reduction in code growth is a result of the smaller average procedure size, which leads to less code being inlined. Code growth is guided by the same heuristics which guide region formation, and further is restricted by a code growth limit of 20% above the original program size, and by the prevention of direct and indirect recursive inlining.

In the case of *clinpack*, code growth was slightly more when partial inlining was used. This unexpected additional code growth can be attributed to the relatively small size of the benchmark, the small number of partial clones created, and the code growth limit being reached at a different time in the compilation. In the other benchmarks, the code growth limit was reached at approximately the same point in compilation as in the full inlining version. Because the procedures that were inlined were smaller on average in the partial inlining version, the result is less code growth before the growth limit was met or exceeded. With *clinpack*, however, an additional instance of inlining was performed that had not been performed in the full inlining version before no other procedures were eligible for inlining.

Runtime performance

The second column of Table 1 shows the percentage change in runtime performance for the benchmarks over the procedure-based strategy, calculated using Equation 1. Performance increase was demonstrated for all benchmarks, ranging from 0.06% to 1.02% for the partial inlining versus full inlining versions. Although the speedups are not dramatic, they demonstrate the benefits for performance that are possible with partial inlining.

For example, the greatest performance increase was seen for *132.ijpeg*, with a 1.02% improvement when using partial inlining versus full inlining. An explanation for this increase is that there were 46 partial clones created for *132.ijpeg*. When fewer partial clones were created, performance improvements were much lower. There were only 3 partial clones created for *clinpack*, resulting in an insignificant 0.06% speedup as compared to the full inlining compilation.

Processor utilization

The third column of Table 2 shows the percentage change in processor utilization for the benchmarks over the procedure-based strategy, calculated using Equation 1. The change in processor utilization for the benchmarks is modest, but in general an improvement. For *clinpack*, processor utilization dropped slightly, by 0.16% for the partial inlining version compared to the full inlining version. This decrease appears to correlate with higher code growth that is not matched by a significant enough increase in perfor-

Table 2. Comparison of compilation unit characteristics by average size and number of units, average profile variance (homogeneity), percentage of invariant units and change to interprocedural scope for the full and partial inlining strategies versus a procedure-based compilation.

Benchmark	Average Unit size full	Average Unit size partial	Number of Comp. Units full	Number of Comp. Units partial	Average Profile variance full	Average Profile variance partial	Percentage Invariant units full	Percentage Invariant units partial	Pct. change in Interproc. scope full	Pct. change in Interproc. scope partial
008.espresso	17.9	18.3	2333	2291	0.361	0.330	90.2	90.3	22.1	21.5
023.eqntott	16.2	17.9	755	740	0.022	0.001	98.6	98.9	27.6	27.0
124.m88ksim	17.7	19.1	4027	4001	0.311	0.240	94.9	95.2	19.8	18.9
130.li	15.6	17.4	2516	2376	0.203	0.193	94.2	94.7	27.6	26.9
132.ijpeg	19.7	22.1	4106	4033	0.284	0.241	93.5	94.8	19.1	17.1
clinpack	18.8	20.7	108	103	1.000	0.600	85.2	87.4	21.6	21.2
average	17.6	19.2	2308	1876	0.364	0.268	97.8	88.6	23.0	22.1

mance. When there is more code to execute due to additional code growth but performance does not improve enough to account for the growth, processor utilization can decrease as a result.

For the remaining benchmarks, processor utilization improved by from 0.06% to 0.30%. These improvements appear to be attributable to better control of code growth with a corresponding improvement in runtime performance.

Unit size

The first two columns of Table 2 shows the average size and number of compilation units (regions) for the benchmarks. The number of regions decreased when partial inlining was performed, due to the effect of partial cloning. When partial cloning is performed, the code is physically moved into a new procedure, leading to the potential for additional demand-driven inlining and region formation within the new procedure. The average region size increased for partial inlining, due to restructuring the program code by grouping code more by execution frequency. This grouping led to slight increases in the amount of inlining done in frequently executed code that was mostly, but not entirely, offset by smaller regions formed in colder portions of the code.

Fewer regions were formed with partial inlining as a result of this partitioning of code into hot and cold regions. Inlining was slightly more likely to be performed in hotter regions than before, leading to slightly more inlining, larger regions on average, and fewer overall regions. Because code growth was less with partial inlining, there was less code to form into regions, as well.

Profile homogeneity

The third and fourth columns of Table 2 shows the profile variance and percentage of invariant code for the full and partial inlining versions. In every case, both profile variance and the percentage of invariant compilation units in a benchmark improved. The improvement in these two characteristics reflects the combination of increased region size and improved performance. When code is better grouped together by execution frequency, profile variance decreases for each compilation unit and therefore overall. As a result of the decreased variance of the profile information in each region, there is an increase in the number of these units that are invariant (i.e., displaying an insignificant amount of variance).

Improved profile variance with an increase in the number of compilation units that are invariant translates to an increase in optimization opportunities for the compiler. With procedure boundaries restructured such that frequently executed code is more often found together, more aggressive optimization may be performed on these regions, trading off the optimization of less frequently executed regions to manage compilation time. The result of better optimization is often improved runtime performance, which is what is seen for these benchmarks when partial inlining is performed, as compared with full inlining.

Interprocedural scope

The last column of Table 2 shows the change to interprocedural scope for the benchmarks, comparing the percentage increase over the procedure-based compilation strategy. In all cases, the use of partial inlining decreased the overall interprocedural scope of the benchmarks as compared with that seen for full inlining. Because of the better grouping by execution frequency that is done with partial cloning, and the additional procedures created as a result, there is a natural decline in the overall percentage of interprocedural code in the benchmarks. This reflects the reality of partial cloning with partial inlining, which by design decreases the amount of code that is inlined.

Discussion

Overall, partial inlining controlled code growth better than full inlining. The increase in the number of procedures led to smaller average procedure size, as was seen in the source level comparison after partial cloning. The reduction in average procedure size could benefit a traditional procedure-based compiler if partial inlining were incorporated.

Modest improvement in execution time was seen

along with a corresponding increase in processor utilization. The one exception to this was *clinpack*; processor utilization actually declined slightly. The change in processor utilization is related to an increase in code size that is not matched by an increase in runtime performance.

The differences in region characteristics can be explained by the improvements seen in profile variance and the percentage of compilation units that are invariant in the program. Less variation of profiling weights within each region indicates that partial inlining can produce better profile homogeneity because it selectively inlines more important segments of code, with regard to the impact on runtime performance, while normally avoiding inlining of less important regions due to partial cloning.

The overall interprocedural scope of the program as measured by the percentage of interprocedural operations decreased with partial inlining. This decrease is actually a beneficial result, occurring because partial inlining produces less code growth since it inlines less code than full inlining. The code that is partially inlined is more desirable, being more frequently executed or hot, while the code that is not inlined, but that normally would be by a full inliner, is less frequently executed or cold code.

4. Related Work

Limited research has addressed partial inlining as a component of other compilation and optimization techniques [1–3, 5], contributing fundamental ideas to our research, although none is directly comparable to our compile-time optimization method. 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 portions, or *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, effectively cloning these sequences.

5. Summary and Future Work

The results of this work indicate that the 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 these modest-sized benchmarks.

The integration of region-based partial inlining into a production-quality compiler and a more comprehensive study of larger programs are planned. Research will also be conducted to explore the use of techniques that improve decision-making when selecting an inlining method, including suppression of cold region inlining, and performing partial cloning.

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.