Demand-driven Inlining Heuristics in a Region-based Optimizing Compiler for ILP Architectures*

Tom Way, Ben Breech, Wei Du and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
email: {way,breech,wei,pollock}@cis.udel.edu

ABSTRACT
While scientists and engineers have been taking on the challenge of converting their applications into parallel programs, compilers are charged with identifying opportunities and transforming codes to exploit the available instruction-level parallelism (ILP) offered by today's uniprocessors, even within a parallel computing environment. Region-based compilation repartitions a program into more desirable compilation units for optimization and ILP scheduling. With region-based compilation, the compiler can control problem size and complexity by controlling region size and contents, and expose interprocedural scheduling and optimization opportunities without interprocedural analysis or large function bodies. However, heuristics play a key role in determining when it is most beneficial to inline functions during region formation. This paper presents and experimentally compares a set of heuristics for demand-driven inlining performed during region formation in terms of static code growth, compile-time memory, and run-time performance.

KEYWORDS
Fine-grain parallelism, optimizing compiler, region-based optimization, inlining

1. Introduction

Interprocedural techniques for program analysis and optimization are increasingly important for fully exploiting modern instruction-level parallel (ILP) architectures. These methods attempt to utilize the higher degrees of available ILP within the context of the growing emphasis on modular applications and object-oriented program design. Region-based compilation [14] for ILP is an approach that enables aggressive interprocedural analysis and program improvement without the expense of performing these potentially costly analyses on large function bodies.

By applying function inlining techniques and constructing new regions to serve as the unit of compiler optimization, region-based compilation obtains more freedom to move code between functions than interprocedural analysis techniques, while maintaining tight control over the compilation unit size and content. In addition to exposing opportunities for interprocedural instruction scheduling and optimization, the size of the resulting regions (i.e., compilation units) has been shown to be smaller than functions, and can be limited to any desired size, in order to reduce the impact of the quadratic algorithmic complexity of the applied optimization transformations [14].

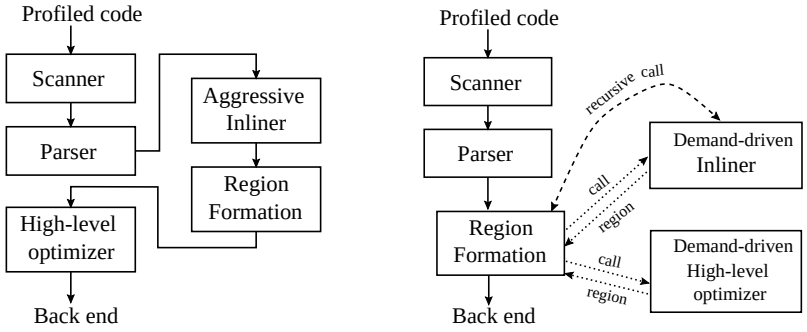
The key component of a region-based compiler is the region formation phase which partitions the program into regions using heuristics with the intent that the optimizer will be invoked with a scope that is limited to a single region at a time. Thus, the quality of the generated code depends greatly upon the ability of the region formation phase to create regions that a global optimizer can effectively transform in isolation for improved instruction-level parallelism. The use of execution profile information in region formation allows the compiler to create regions that more accurately reflect the dynamic behavior of the program. With profile-based partitioning, an optimizer can apply expensive, sophisticated optimization analysis to high execution frequency regions, and little to no optimization on low execution frequency regions.

Function inlining plays an important role in enabling optimizations that cross function boundaries by making it possible for the region formation phase to easily analyze across these boundaries and form interprocedural regions, which consist of instructions from more than one function of the original program. However, too much inlining leads to excessive code growth, while too little inlining, or poorly selected inlining, leads to less gain in run-time performance.

Figure 1 depicts two region-based compilation frameworks, indicating the role of function inlining. The region formation phase of Hank et al. [14] is designed to run on a form of the whole program which has been aggressively inlined in order to remove function call barriers during region formation. In previous research, we described how the potentially costly aggressive inlining step can be replaced by a recursive region formation algorithm that incorporates a demand-driven inliner [23].

This paper presents a set of inlining heuristics for demand-driven inlining directed by region formation. In particular, two heuristics guide inlining: (1) A first-order heuristic determines the order in which functions are con-

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



(a) with aggressive inliner (b) with demand-driven inliner

Figure 1. Compiler phases of two region-based frameworks.

sidered for region formation, and (2) A second-order heuristic determines whether or not a given function is to be inlined when a call site is reached during region formation. We have developed and implemented heuristics for both first-order and second-order inlining within the Trimaran ILP compiler [19]. We present the results of an experimental study of different combinations of these heuristics, comparing their effectiveness as measured by their impact on compilation and run-time performance.

2. Demand-driven Inlining Heuristics

Function inlining has been a widely researched and accepted means to enable whole program optimization [4, 5, 10, 16, 17, 18, 20, 21]. In function inlining, function calls are physically replaced with the complete body of the called function. Heuristics for deciding the order to consider call sites and whether a given call site should be inlined have the goal of achieving a reasonable tradeoff between the benefits and pitfalls of inlining, given some maximum limit set for acceptable code growth. Heuristics can be based on static program information, run-time profiling information, or a combination [3, 22].

The design of inlining heuristics must address several challenging issues: (1) correctness, maintaining semantic equivalence with the original code, (2) code growth, ensuring that the program size does not grow too large for memory, (3) compilation time, realizing that large functions created by inlining will increase optimization and code generation time, (4) run-time performance, noting that some call sites will lead to more opportunity for increased optimization than others, and (5) handling of hazards to inlining such as recursion, where you might not want to prohibit inlining completely, but need to be concerned with correctness and code growth.

Function inlining has been shown to improve compiler analysis and optimization [2, 6], register usage, code locality and execution speed [6], data flow analysis used to generate more efficient code in the callee, *specialized* to its calling context [2, 3], and to enable easier constant propagation and elimination of redundant operations [6]. However, inlining can increase register pressure [3, 6, 9], code size [3, 6], instruction cache misses [2, 3, 6], and compilation time.

In the remainder of this section, we describe the heuristics that we have considered for demand-driven inlining within region formation. *First-order heuristics* select the order to consider functions for region formation, which will affect the order of demand-driven inlining decisions. Because demand-driven inlining within a given function is considered at call sites as region formation is performed for that function, the order of decisions for demand-driven inlining follows the flow-directed manner in which region formation is done over a given function's control flow graph. *Second-order heuristics* determine whether or not a particular callee is inlined at a call site during region formation.

2.1 First-order Heuristics

The first-order heuristics we have studied attempt to order functions from most to least important in terms of optimization opportunity. In particular, we examined three possible first-order heuristics for demand-driven inlining. The most precise measurement of function importance is actual dynamic run-time profiling which comes at the cost of an initial instrumentation, compilation and execution. Functions are ordered from highest to lowest percentage of overall run-time spent in the function, based on profiling information.

Static estimates of importance provide less costly heuristics, but also tend to produce less precise information. One heuristic based on static estimates orders functions from most to least number of static call sites within the function, and within that order from smallest to largest function size. More importance is assigned to functions with the highest percentage of call sites compared with code size. This increases the chance that region formation will be performed interprocedurally, producing more scheduling and optimization opportunities, while control-

ling code growth by considering smaller functions before larger ones.

Another ordering we considered is based on the *loop call weight* of a function, assigning more importance to functions which contain more call sites within loops, and increased importance for those call sites that are more deeply nested. The loop call weight is computed as: $\sum_{i=1}^{n} loopdepth_i \times W$, where n is the number of call sites in a function, and W is the loop depth weight constant. A value of 10 is used for W to assign an order of magnitude increase in significance to successive loop depths, since, intuitively, interior loops consume more execution cycles than do their enclosing loops.

2.2 Second-order Heuristics

The second-order heuristics attempt to increase instruction scheduling and optimization opportunities while minimizing code growth. For correctness, functions where there are mismatches in the number and types of parameters between the call site and callee are not inlined. Similarly, we prevent inlining when the compiler determines that memory regions associated with arguments to a function may overlap, using a conservative static approach that includes preventing inlining when any arguments are pointers.

To avoid high code growth, we prevent inlining once the the overall code size has increased more than 20% percent above the original size. A code growth limit of 20% has been shown to minimize unnecessary code growth while still allowing beneficial inlining [14].

Similarly, we prevent inlining of functions that are directly or indirectly recursive. While a limited amount of recursive inlining can produce a result analogous to loop unrolling, the potential for exponential code growth is high.

Functions that are more frequently executed than a fixed frequency or with some desired ratio over the frequency of the caller are inlined. Region formation already uses this second-order heuristic, such that inlined functions will always be executed at least 50% as frequently as the seed block of their enclosing region.

Only functions that are less than a static maximum size are inlined to limit code growth, and functions with higher call overhead compared with their code size are inlined.

3. Experimental Study

3.1 Methodology

Implementation of our techniques and our experiments has been conducted in the context of the Trimaran compiler, an integrated ILP compilation and performance monitoring infrastructure [19]. Hank et al.'s region formation [14] was already implemented in Trimaran. We integrated our demand-driven region formation algorithm, along with an inliner that works at a lower code level than previously implemented. We extended the demand-driven inliner with a number of new second-order inlining heuristics. The existing region formation phase was enhanced to encapsulate regions into functions so that region-based optimization could be performed, and to incorporate additional first-order inlining heuristics.

Benchmark	Lines of source code	Num. of func.	Trimaran Lcode instructions		
			Total prog. size	Max. func. size	Avg. func. size
124.m88ksim	19092	252	55783	1537	192.7
130.li	7597	357	31552	526	80.9
023.eqntott	3628	62	11738	1507	168.7
026.compress	1503	16	2601	884	151.8
paraffins	388	10	1115	324	104.3
clinpack	313	9	1294	585	182.9
bmm	106	7	538	97	72.3
fact3	48	4	220	79	53.0

Table 1. Benchmark size and function characteristics.

We ran our experiments on a suite of eight benchmarks, four from SPEC (124.m88ksim, 130.li, 023.eqntott, and 026.compress), one from Netlib (clinpack) and three that were included in the Trimaran distribution (paraffins, bmm and fact3). Table 1 indicates the number of source code lines and function definitions for each benchmark, as well as the number of *Lcode instructions*. *Lcode* is a low-level intermediate code roughly equivalent to *three-address code*.

We investigated several promising combinations of first-order and second-order inlining heuristics against a baseline compilation with no inlining, **HØ**. Table 2 describes the implemented and evaluated combinations. The heuristic combinations use various static and dynamic esti-

Heur.	Description
H1	Original region-based compilation. Aggressive inlining with standard code growth limit, then region formation; first- and second-order inlining heuristics as defined by [14]. **1st-order:** Run-time profile ordering. **2nd-order:** Inlined functions guaranteed to be executed at least 50% as much as seed block in their region.
H2	Demand-driven inlining with simple static heuristics. **1st-order:** From most to least number of static call sites, and smallest to largest function size. **2nd-order:** Same as H1, plus only inline if callee size $\leq$ 25 [13].
H3	Demand-driven inlining with simple static heuristics. **1st-order:** Same as H2. **2nd-order:** Same as H1, plus prevent inlining direct or indirect recursion. (Increases number of functions into which inlining is performed)
H4	Based on loop call weight. **1st-order:** Order by decreasing loop call weight, and then smallest to largest function size **2nd-order:** Same as H3
H5	Based on execution cycles. **1st-order:** From most to fewest execution cycles, then smallest to largest function size. **2nd-order:** Same as H3.
H6	Based on execution cycles and loop call weight. **1st-order:** Same as H5. **2nd-order:** Minimum loop call weight of 10 to inline. (Only inline if contains at least 1 call within at least one loop)

Table 2. Heuristic Combinations.

mations to affect inline ordering and decision making.

H1 is Hank et al.'s method without modification. **H2** and **H3** replace **H1**'s 1st-order heuristic with static measurements, assigning higher priority to functions which have more call sites compared to code size, a fast estimator of greater potential for interprocedural optimization balanced with code growth control.

The 2nd-order heuristic in **H1** is extended to control code growth in **H2** with a threshold that limits the size of functions being inlined, and in **H3** by preventing any recursive inlining. This same recursion prevention is used in **H4**, while a more accurate static estimator, loop call weight, is used in the 1st-order heuristic to predict profiling weight.

Heuristics **H5** and **H6** rely on runtime profile information for 1st-order decisions, handling first the functions which comprise more execution time with less impact on code size by taking first the functions with more execution cycles as compared to code size. For 2nd-order heuristics, **H5** uses **H3**'s recursion prevention, and **H6** uses a threshold based on loop call weight, which limits inlining to functions that have a least one loop and therefore potentially more execution cycles than a function with no loops.

A static code growth limit of 20% was used for all experiments [16]. For all heuristics, inlining was prevented in standard ways for parameter type and number mismatch, and memory region overlaps in parameters to ensure correctness. After region formation, global optimization passes were applied to the individual regions created by the different region formation strategies. Execution statistics were gathered by running the resulting code through the Trimaran backend on a simulated 4-wide EPIC (**E**xplicitly **P**arallel **I**nstruction **C**omputer) machine with 128k I-cache (12 cycle miss latency), 128k D-cache, 1Mb L2-cache (4 cycle latency), and a 30 cycle memory latency. Execution timings where measured in simulated cycles of execution.

We compared the heuristic combinations by measuring four effects in terms of the percentage change of each combination versus **HØ**, computed as: *((***HØ** - *Heuristic)/* **HØ***) * 100*. In particular, we evaluated: (1) **code growth**, (2) **compilation time**, including the time to compile the source code up through region formation and region-based optimization, (3) **operations per cycle**, which measured how well the scheduler was able to make use of available processor resources, and (4) **execution time** which measured more directly the impact of inlining heuristics on region formation and region-based optimization, performance improvement being a primary goal of any compiler optimization technique.

3.2 Results and Discussion

The results of our experiments are presented in Tables 3, 4, 5 and 6. For the cases that cause more code growth, execution time also improves. The more naive heuristics of **H2** lead to the smallest increases in code size and compilation time, but also do not improve performance as much

	Code growth					
Benchmark	H1	H2	H3	H4	H5	H6
124.m88ksim	7.9	1.1	16.0	14.8	11.9	11.9
130.li	8.4	0.4	7.7	5.9	4.2	4.2
023.eqntott	6.5	0.2	17.8	17.8	15.2	15.2
026.compress	17.3	0.0	23.1	23.1	21.1	21.1
paraffins	30.8	0.0	36.8	33.0	13.3	13.3
clinpack	15.0	3.1	15.4	15.4	15.5	15.5
bmm	21.5	0.0	21.6	18.5	23.3	23.3
fact3	12.0	6.7	17.4	17.4	17.4	17.4
average	14.9	1.4	19.5	18.2	15.2	15.2

Table 3. Percentage change in code growth over HØ.

	Compilation time					
Benchmark	H1	H2	H3	H4	H5	H6
124.m88ksim	4.0	-2.1	27.6	18.4	27.6	27.4
130.li	4.5	0.0	26.8	24.5	25.6	24.9
023.eqntott	1.8	-0.1	2.4	4.8	9.3	7.9
026.compress	-8.3	-1.3	-2.8	0.0	5.6	2.8
paraffins	4.8	-4.7	4.8	9.5	2.0	2.0
clinpack	5.6	0.0	27.2	27.2	27.2	23.0
bmm	-15.8	-1.5	-5.3	-5.3	-5.3	-5.3
fact3	0.0	7.7	15.4	15.4	15.4	15.4
average	-0.4	-0.3	12.0	11.8	13.4	12.3

Table 4. Percentage change in compilation time over HØ.

	Ops per cycle					
Benchmark	H1	H2	H3	H4	H5	H6
124.m88ksim	-1.50	-0.50	-0.50	0.00	0.00	0.00
130.li	3.53	0.00	-1.77	-4.70	-4.75	-4.75
023.eqntott	1.89	-1.88	1.89	3.77	1.89	1.89
026.compress	-5.37	0.00	-5.37	-5.37	-5.37	-5.37
paraffins	2.67	0.00	0.00	2.67	3.21	3.21
clinpack	-0.90	0.00	-0.90	-3.63	-3.63	-3.63
bmm	0.00	0.00	0.96	0.00	0.00	0.00
fact3	0.00	0.00	8.00	8.00	8.00	8.00
average	0.04	-0.30	0.29	0.09	-0.08	-0.08

Table 5. Percentage change in ops/cycle over HØ.

	Execution time					
Benchmark	H1	H2	H3	H4	H5	H6
124.m88ksim	-6.13	-1.31	-3.90	-9.22	-9.13	-9.13
130.li	-8.49	-2.16	-4.01	-12.53	-12.20	-12.20
023.eqntott	-3.17	1.31	-4.02	-6.11	-1.90	-1.90
026.compress	-3.11	0.00	-3.11	-3.11	-2.98	-2.98
paraffins	-3.26	0.02	-0.19	-3.71	-3.50	-3.50
clinpack	-4.83	0.00	4.26	-5.35	-5.35	-5.35
bmm	0.10	0.00	-0.08	-0.23	-0.16	-0.16
fact3	-12.59	-8.39	-16.02	-16.02	-16.02	-16.02
average	-5.19	-1.32	-3.41	-7.04	-6.41	-6.41

Table 6. Percentage change in execution time over HØ.

as the other more sophisticated methods. Larger increases in code growth and compilation time do not always translate to improvements in processor utilization and execution speed, indicating that bounding code growth is important.

The more scientific codes (124.m88ksim, 023.eqntott, paraffins, clinpack), tend to benefit the most from increases in code growth and compilation time (which is also optimization and scheduling time) in terms of their processor utilization and execution time, particularly for the most advanced profile-estimating (**H4**) and profile-based (**H5** & **H6**) methods. Small benchmarks (bmm, fact3), by both size and number of functions, are less predictable, although significant performance gains are seen with **H3-H6**, with most showing improvement over the original region-based technique (**H1**). Benchmarks with more recursion (130.li, 026.compress, fact3) required more compilation time and gained comparatively less in performance improvements than the others.

The combination of heuristics in **H4** proved consistently to be the most effective at controlling code growth and compilation time while improving run-time performance. The fact that **H4** bases inlining decisions on the static loop call weight, which estimates run-time behavior, rather than the actual profiling information itself, as in **H5** and **H6**, is significant, indicating that profiling may not be necessary for making good demand-driven inlining decisions during region formation. Profiling generally is more precise than static estimates because it directly measures program behavior at run-time, but requires more overhead and depends on the data used for the profiling.

4. Related Work

Procedure inlining research is generally the study of heuristics to guide inlining decisions, since inline substitution is a well-understood and basic compilation technique. The problem of determining the optimum set of inlines (or clones) is NP Complete [8]. Thus, extensive research into inlining heuristics and the factors that bolster or limit its effectiveness has been performed [1, 4, 5, 7, 10, 13, 16, 17, 18, 20, 21].

Empirical evaluation has revealed that a hybrid inlining policy that considers inlining either the original and current versions of a procedure outperforms either of the two policies alone [18]. It was determined that a greedy strategy where the original version of a procedure is used at all inlining steps can be trapped at a local maxima, and does not produce better results that the other methods.

There is strong experimental evidence that the incorporation of profiling information into inlining heuristics to select most important call sites first is crucial, with inlining functions ordered most to least frequent [5, 8, 16] the most common technique. Another variation on using profiling information is to inline functions when their invocation time is longer than function execution time [8].

Performance also can improve when the compiler can anticipate the enabling effects of inlining, such as shrinking optimizations like constant folding through estimation of the resulting benefits [2], by trial inlining [11], and when it enables high-payoff optimizations [13]. In a dynamic optimizing Java compiler, the use of a cost-based inlining heuristic, using a node-weight scheme on both the static and dynamic call graph, has been shown to be beneficial to execution time [3].

The majority of inlining-related research has focused on developing and evaluating heuristics within a procedure-oriented compilation system where the unit of compilation is a procedure, as opposed to a region-based compiler where the unit of optimization is selected and controlled by the compiler, and inlining plays a key role in region formation. In addition, the techniques were applied as a separate pass, typically early in compilation, rather than in a demand-driven setting within a phase of optimization.

Demand-driven analysis [12, 15] has the goal of applying analysis and other optimizations only when they are needed, on demand, so that compile time and memory requirements can be managed. For example, if profiling information is being used during compilation, the compiler can decide on the fly which procedures it wishes, for example, to inline, as it reaches them during the compilation process. In this way, optimization can be prioritized and customized based on characteristics of different sections of code, and compilation time can be better controlled.

5. Conclusions and Future Work

This paper presents the first experimental study of potential heuristics for demand-driven inlining, which plays a key role in increasing ILP optimizations within a region-based compiler. Because inlining is performed on demand during region formation, the problem is more complex than a separate inlining pass early in compilation. Based on our set of benchmarks which includes different sizes, scientific as well as non-scientific codes, and different degrees of recursion, we have discovered that heuristics based on static information can achieve as good or even better runtime performance improvements with a region-based compiler than heuristics based on profiling information. We plan to continue this work, focusing on the design, integration, and evaluation of partial inlining within the region-based optimizing compiler framework.

References

[1] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 241–249, 1988.

[2] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and dynamic heuristics

for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000)*, pages 52–64, Jan. 2000.

[4] J. M. Ashley. The effectiveness of flow analysis for inlining. *ACM SIGPLAN International Conference on Functional Programming*, pages 99–111, 1997.

[5] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *Computing Systems*, 26(4):345–420, 1994.

[7] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[9] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software Practice and Experience*, 18(8):775–790, Aug. 1988.

[10] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18:89–101, 1992.

[11] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 273–282, 1994.

[12] E. Duesterwald. *A Demand Driven Approach for Efficient Interprocedural Data Flow Analysis*. PhD thesis, University of Pittsburgh, 1996.

[13] M. W. Hall. *Managing Interprocedural Optimization*. Ph.D. thesis, Rice University, Apr. 1991.

[14] 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.

[15] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 104–115, 1995.

[16] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257, 1989.

[17] S. Jagannathan and A. Wright. Flow-directed inlining. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, 1996.

[18] O. Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24:55–72, 1998.

[19] 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.

[20] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[21] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, Sept. 1977.

[22] M. Serrano. Inline expansion: when and how? In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 143–157, Southampton, UK, Sept. 1997.

[23] 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.