Procedure Boundary Elimination for EPIC Compilers

Spyridon Triantafyllis Manish Vachharajani David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544
{strianta, manishv, august}@cs.princeton.edu

Abstract

Procedures are the basic units of compilation in traditional optimization frameworks. This presents problems to compilers targeting EPIC architectures, since the limited scope of a single procedure is usually insufficient for extracting ILP and identifying enough optimization opportunities. Although inlining can expand the scope of optimization routines, it is not applicable to all call sites and can cause excessive code growth, which can in turn adversely affect cache performance and compile-time resource usage.

In this paper we propose a novel compilation strategy called Procedure Boundary Elimination (PBE). PBE unifies the whole program into a single compilation unit, which is then restructured into units better suited to optimization than the original procedures. A targeted specialization phase exposes further optimization opportunities while limiting code growth only to the cases where it is beneficial. Unlike inlining, PBE can eliminate all procedure calls while avoiding the cost of excessive code growth.

1. Introduction

Achieving good performance on a modern wide-issue architecture is critically dependent on compiler support. Apart from traditional code simplification and redundancy elimination optimization routines, a modern aggressively optimizing compiler has to efficiently exploit complex computational resources, expose instruction-level parallelism (ILP), and avoid performance pitfalls such as memory stalls and branch misprediction penalties. The dependence of performance on compiler quality is especially pronounced on EPIC architectures, since these architectures require compiler-constructed explicit schedules.

In order to meet the challenges posed by EPIC architectures, a compiler has to rely on a rich set of aggressive optimization and analysis routines. The ability of such routines to produce efficient code can be greatly hampered by the traditional procedure-based compilation approach. This is because the original breakup of a program into procedures serves software engineering rather than optimization goals, and thus individual procedures may not present the best scope to optimization and analysis. For example, procedure calls within loops can conceal cyclic code from the compiler, and breaking up a task into too many small procedures may prevent a scheduling routine from constructing traces long enough to provide sufficient ILP opportunities. Modern software engineering techniques such as object-oriented programming, which typically encourage small procedures and frequent procedure calls, exacerbate the problem.

To alleviate the effects of inconveniently placed procedure boundaries, traditional compilers have employed interprocedural analysis and aggressive inlining. Interprocedural analysis exposes more information to the optimizer and can be employed as extensively as compile time permits. However, designing interprocedural analysis routines is complicated by the fact that such routines have to take into account parameter-passing mechanisms and other calling conventions. Inlining ([1],[2],[3]), originally proposed to limit call overhead, copies the body of a callee procedure into the body of the caller. This not only exposes more code to analysis routines, but also allows subsequent optimization routines to specialize the code of the callee for each particular call site. Unfortunately, the benefits of aggressive inlining come at the cost of extensive code growth. This can lead to poor instruction cache performance, as well as slow down the compilation process. Since the adverse effects of code growth can very quickly become prohibitive, inlining is usually limited to frequently executed call sites with relatively small callees. The applicability of inlining is further limited by its inability to handle recursive and virtual procedure calls. Inlining is therefore only a partial solution to the optimization scope problem.

This paper proposes a novel compilation strategy, called Procedure Boundary Elimination (PBE), to overcome the limitations of traditional procedure-based compilation. The PBE approach first eliminates procedure boundaries without causing any code growth. The program is then partitioned into *regions* [4] in order to present to subsequent optimization routines a set of compilation units that are both manageable in size and adequate in scope. Targeted code specialization is then applied in order to create more optimization opportunities, while limiting code growth only where it is likely to produce significant benefits. Standard optimization and analysis routines, with some essential modifications, are subsequently applied to each region. In this way PBE offers increased optimization scope and specialization opportunities, while keeping code growth in check. Unlike inlining, PBE handles all call sites in a uniform way, regardless of recursive cycles or callee size. An added benefit of PBE is that the design of global analysis routines is simplified, since such routines do not have to navigate around procedure calls and platform-specific calling conventions.

2. The Optimization Scope Problem

In a traditional compiler each procedure is treated as a separate compilation unit. Since procedures are defined by the programmer according to software engineering considerations, they may not be ideally suited for achieving maximum optimization efficiency. In modern EPIC architectures, where the dependence of performance on aggressive optimization is especially pronounced, limiting the scope of optimization and analysis routines to a single procedure may lead to significant performance degradation. Section 2.1 analyzes this problem in more detail.

To address the problem of performance degradation due to inconveniently placed procedure boundaries, most modern compilers employ aggressive inlining. Although inlining can lead to large performance gains, it is a solution of limited applicability and effectiveness. Section 2.2 presents inlining and explains the need for a more general solution to the problem of optimization scope.

2.1. Problems with Procedure-Based Compilation

In modern software systems the breakup of a large program into smaller, more manageable procedures serves two unrelated and ultimately contradictory purposes. As a programming unit a procedure must be small, elegantly written, and conceptually coherent. As a unit of optimization and analysis a procedure must be large enough to provide an adequately wide scope for optimization and analysis, and should ideally contain pieces of code that are strongly correlated, both in the sense that they usually execute together and in the sense that they operate on common data.

The reason why these two roles may be contradictory is best illustrated through an example. In Figure 1a we can see a small procedure `f` with two arguments, which is called from two different locations: one is in basic block `C`, which is part of a loop `L`, and one is in the less frequently executed block `K`. Hot basic blocks are shown with a thicker border. In the following discussion we will assume that these blocks, that is `B`, `C`, `D`, `E`, `F` and `H`, execute much more frequently than the rest. As for the small pieces of code within the blocks, let `LI1` and `LI2` be two instructions that are invariant with respect to the loop formed by blocks `B`, `C` and `D` Also, let `f(rx, ry)` be an abbreviation for making a call to procedure `f` with parameters `rx` and `ry`; depending on the calling convention, this may correspond to a multiple instruction sequence, including instructions that copy the parameters to specified locations, as well as instructions that save and restore caller-saved registers.

Putting the blocks `E`, `F`, `G` and `H` into a separate procedure may be a good decision from a software engineering perspective. As we can see in Figure 1a, this allows the code of these four blocks to be reused in block `K`. The code in these four blocks may also be conceptually different from that in loop `L`. However, partitioning the code in this way conceals optimization opportunities. Although the code of `f` is frequently executed as part of `L`, it is not part of the loop as far as optimization and analysis are concerned. Thus, loop unrolling or software pipelining would benefit only the three blocks in `L`, despite the fact that executing code in `f` probably accounts for a significant portion of the time spent in `L` during program execution. Moreover, instructions in blocks `C` and `D` cannot be scheduled together. This may limit the ability of the scheduler to exploit instruction-level parallelism.

Classical optimization routines can also suffer from the inconveniently placed procedure boundaries in Figure 1a. For example, although parameter `b` in `f` has the constant value 5, no constant propagation is possible. That's because most dataflow analysis routines operate on the procedure level, and would therefore miss this fact. Also, instruction `LI2` cannot be moved out of the loop `L`, even though it is loop invariant.

Various efforts to remedy this situation have been made in the past. One such effort is to generalize certain analysis routines, so that they can take into account interprocedural information. In the example in Figure 1a, an interprocedural dataflow analysis routine might be able to detect the opportunities for constant propagation in parameter `b`. Although interprocedural analysis can be very useful, it is usually too expensive to be applied extensively.

A much more systematic effort to solve the problems caused by procedure boundaries is aggressive inlining. The benefits and drawbacks of aggressive inlining are presented in the next section.

2.2. Inlining: An incomplete solution

Inlining eliminates inconveniently placed procedure boundaries by duplicating the callee's code into the call site. Although the original purpose of inlining was to eliminate call overhead, in today's optimizing compilers inlining is used aggressively in order to increase the scope of optimization. By making the code of the callee visible to its caller, new optimization opportunities can be identified. At the same time important call sites acquire a private copy of the callee's code, which can then be specialized in that particular context. Inlining has been extensively studied in the literature ([1], [2], [3], [5], [6]).

The small piece of code shown in Figure 1a can be transformed by inlining to the equivalent piece of code shown in Figure 1b. Here the code of `f` has been copied inside the loop `L`. In addition to eliminating the frequently incurred call overhead in block `C`, this creates new optimization opportunities.

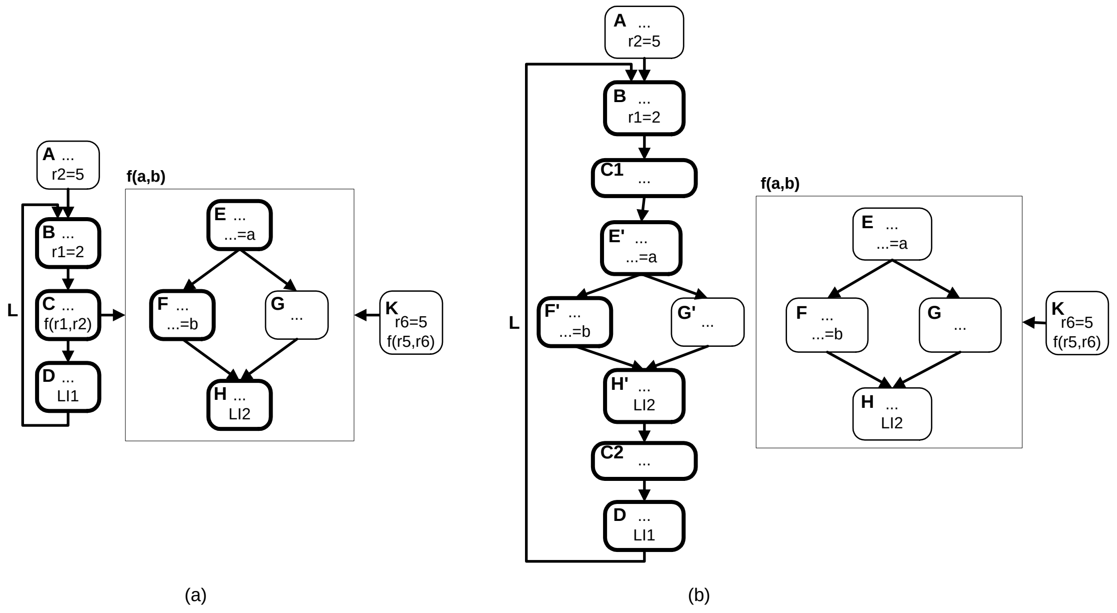

Figure 1: A small procedure with two call sites, (a) before and (b) after inlining

Now instruction `LI2` can be moved out of the loop, and constant propagation can be performed on both `a` and `b`. Loop optimization and scheduling routines can now operate on the whole body of the loop, instead of being limited to blocks `B`, `C`, and `D`.

Unfortunately, the benefits of aggressive inlining come at the cost of extensive code growth. For example, in Figure 1b the body of loop `L` has grown from 3 to 7 basic blocks. Such code growth can adversely affect instruction cache performance, often negating performance benefits due to new optimization opportunities. Moreover, a modern compiler includes many routines of superlinear complexity. When presented with larger procedures as a result of extensive inlining, a compiler's time and memory usage during optimization may grow excessively.

Experimental results found in the literature verify the above observations. For example, the inliner presented in [2] causes an average performance improvement of 11% at the cost of 17% average code growth on a set of eight benchmarks, and [4] reports an increase by 8.3 times in the optimization time of the Perl benchmark when 20% of call sites are inlined.

Code growth concerns limit the applicability of inlining to frequently executed call sites with small callees. Moreover, the steep increase in compile time due to aggressive inlinining forces most compilers to make conservative inlining choices in order to keep optimization tractable. Finally, inlining cannot handle recursive functions properly. Although recursive cycles are in essence loops, inlining cannot expose the looping behavior of recursive functions, which would allow the optimizer to apply loop optimizations to recursive function bodies. Instead, inlining can only eliminate the first stages of recursion, but must ultimately leave the recursive call in place.

The drawbacks of inlining as a solution to the optimization scope problem stem to a great extent from the fact that it was not designed to solve this particular problem. Indeed, although inlining alleviates some of the problems caused by procedure boundaries, it is itself constrained by those boundaries. That is because inlining operates on the procedure level: it can respond to optimization scope problems only by duplicating whole procedures. This leads, among other things, to excessive code growth. In the example in Figure 1b, the whole body of procedure `f` is duplicated, when most optimization benefits are likely to come only from the "hot" path, `E`→`F`→`H`. Partial inlining ([5], [6]) has been proposed to address this problem. However, the freedom of partial inlining to specialize code is still constrained by the original procedure boundaries, the quality of its results depend on the structure of the callee, and it is still not applicable to recursive, virtual, and certain large callees. Thus partial inlining is an effort to alleviate the drawbacks of inlining, rather than an effort to eliminate these drawbacks.

3. Procedure Boundary Elimination

As discussed in section 2, optimization and analysis routines in compilers for modern architectures can suffer from limited optimization scope. Although inlining can extend the scope of optimization routines by copying callee procedures to their call sites, its benefits are offset by significant drawbacks, including excessive code growth and limited applicability.

Our goal in this section is to define a code transformation that eliminates problems caused by inconveniently placed procedure boundaries without causing unnecessary code growth. Such a transformation would obviate the need for inlining, while at the same time exposing more optimization opportunities and exhibiting better compile-time behavior than inlining. Although some code growth will be allowed, code duplication will only happen when it is deemed beneficial for optimization, and not for the purpose of eliminating procedure calls. We call the proposed method Procedure Boundary Elimination (PBE).

PBE involves three separate phases. The first phase is *procedure unification*, which transforms the whole program into one single compilation unit. In the second phase, *region formation*, this single compilation unit is repartitioned in more manageable pieces using the algorithm proposed in [4]. In the third phase, *targeted code specialization*, selected parts of the program are duplicated in order to provide code specialization opportunities. Apart from these three phases PBE also requires certain modifications in existing optimization and analysis routines in order to be effective.

Procedure unification, region formation and targeted code specialization are covered in Sections 3.1, 3.2, and 3.4 respectively. Required modifications in existing optimization and analysis routines are covered in Section 3.3.

3.1. Procedure Unification

The first step in PBE, called Procedure Unification, is to eliminate procedure boundaries by replacing call and return instructions with normal branches. Thus the whole program effectively becomes a single procedure. As a result, optimization and analysis routines can operate on the widest possible scope. To return to the example in figure 1, the original code will be transformed by procedure unification to the code in figure 2a.

Apart from replacing calls and returns with branches, procedure unification must also take care of the rest of the semantics of a procedure call. These include parameter passing and stack frame setup, as well as saving and restoring caller- and callee-saved registers. We will first present the solution to these problems for nonrecursive procedures, and then we will adapt our solutions to the recursive case.

3.1.1. Parameter passing

PBE is applied early on in the optimization process, before register allocation. Parameter passing can therefore be performed by creating a new virtual register for each parameter of each procedure. Before branching into a (former) procedure body, a piece of code needs to move the procedure's (former) parameters into the virtual registers designated for that procedure. Parameters that are too big to fit into registers can be moved into designated memory locations. Parameter passing for recursive calls requires some extra complication, as discussed in section 3.1.4.

3.1.2. Stack frame setup

Since we are dealing with nonrecursive procedures, there is no need to maintain a stack for activation frames. Instead, the activation frame of each procedure can occupy a constant memory address range into the global variable space. This can be achieved by simply assigning a separate memory address range for the activation frame of each procedure. A more sophisticated implementation can avoid wasting memory space by assigning overlapping memory spaces to procedure stack frames. This is possible, since only procedures that can be active at the same time need non-overlapping stack frames. An interference graph between procedures needs to be created, where two procedures are considered interfering if they can be active at the same time. A graph coloring algorithm can then be used to assign address ranges to activation frames in a near-optimal way.

3.1.3. Handling caller-saved and callee-saved registers

Since we are dealing with nonrecursive procedures, we can just rename virtual registers so that each procedure uses a disjoint range of virtual registers. The register allocation routine can then find an optimal allocation of these virtual registers to actual machine registers.

Forcing each procedure to use a distinct set of virtual registers may cause a huge increase in the number of virtual registers used in a program. However, the increase in register pressure will be much more modest. This is because only virtual registers in procedures that can be active together can interfere with each other. Even then, live range splitting should be able to reduce register pressure in most cases. In the example in figure 2a, a virtual register that is defined in block `B` and used in block `D` increases register pressure in the former body of `f`. However, since such a register is just "live through" blocks `E`, `F`, `G`, and `H`, its live range can be split just before the branch to `E` and restored just after the branch back to `C2`. In the worst case, live range splitting will have to insert as many saves and restores as the original calling convention. However, in most cases live range splitting will be able to exploit its extra degrees of freedom in order to make much better decisions than the original calling convention.

3.1.4. Dealing with recursive procedures

By "recursive procedure" we mean any procedure that participates in a cycle in the original call graph of a program. This covers both simple and mutual recursion. Such procedures can be easily identified before procedure unification.

It is ultimately impossible to implement the semantics of recursive procedure calls without using a stack. Therefore the solutions discussed above for parameter passing, activation frame setup and saving registers are not applicable to recursive

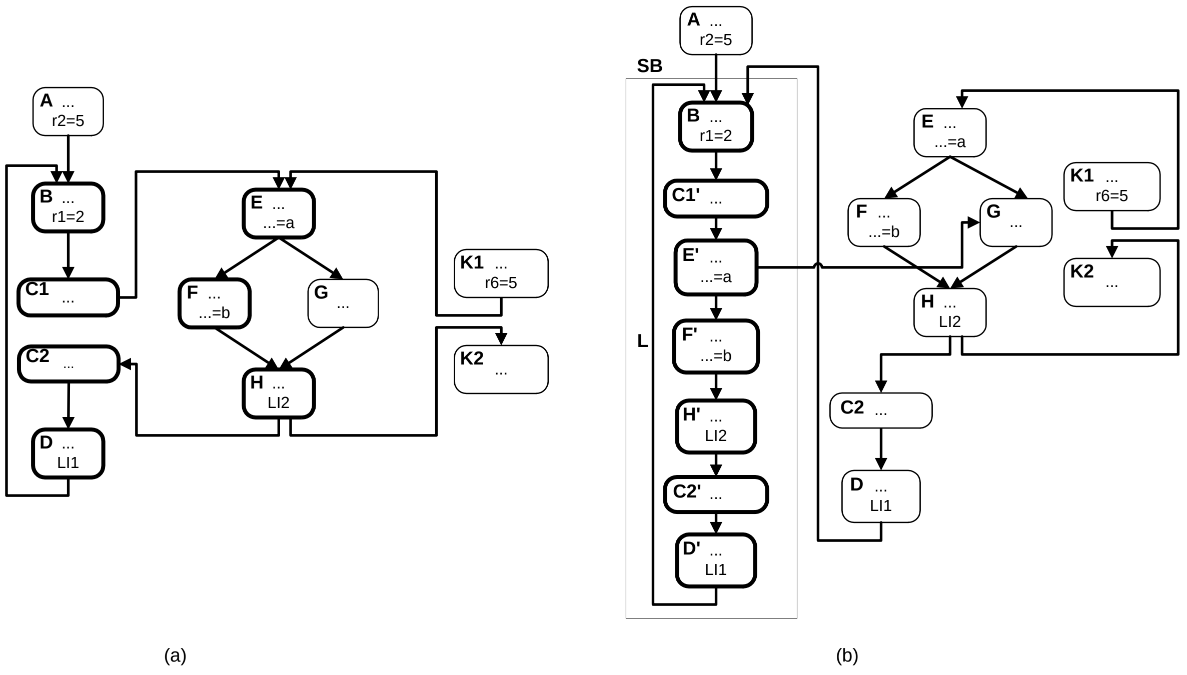

Figure 2: A small procedure with two call sites after (a) procedure unification and (b) procedure unification and superblock formation

procedures. Even after procedure unification, recursive procedures will have to allocate their parameters and local variables in the stack, and save and restore any registers they modify around recursive procedure calls.

However, recursive procedures can also benefit from procedure unification. After procedure unification, recursive call sequences will be transformed to loops. Traditional optimizations can then be applied on those loops. Such optimizations can take advantage of loop invariant code motion and copy propagation in order to reduce the number of registers used within the recursive loop, and therefore the amount of data that has to be saved on the stack.

3.2. Region formation

The result of procedure unification is a single, huge compilation unit. Since aggressive optimizers incorporate many superlinear optimization and analysis routines, a compilation unit of that size is bound to cause an explosion in time and memory usage during optimization. Therefore, PBE cannot be practical unless this compilation unit can be repartitioned into more manageable pieces.

Fortunately a method for partitioning large compilation units into smaller, more manageable pieces has already been proposed. This method, called *region formation*, has been described in detail in [4]. Region formation has been originally developed in order to cope with the compile-time explosion caused by over-aggressive inlining, but it becomes even more valuable in the context of PBE.

Region formation uses profile information in order to break up a large compilation unit into *regions*, i.e. pieces of code that usually execute together. These regions are subsequently used as units of compilation. Experimental results in [4] demonstrate that region formation can keep optimization time constant in the presence of ever-growing procedure size, at the cost of insignificant performance penalties at runtime. Although these experimental results refer to inlining, there is no reason to assume that region formation cannot be just as effective in the context of PBE.

3.3. Modifications in existing optimization and analysis routines

Expanding the scope of optimization and analysis routines cannot be accomplished by simply eliminating procedure boundaries. This is because procedure unification introduces irregular program structures that existing compiler routines may not be able to deal with.

For example, as shown in figure 2a, the natural loop `L` is lost after procedure unification. That is because blocks `E`, `F`, `G`, and `H` are not dominated by the loop's head, `B`. Moreover, a traditional dominator analysis routine would conclude that blocks `C2` and `D` can be reached through the path `K1`→`E`→`F`→`H`→`C2`→`D`, although this path is in fact never taken. Therefore loop optimizations are not available on these blocks. For example, although the loop-invariant instruction `LI1` can be safely moved in block `A`, traditional analysis routines would conclude that such a move is not legal.

The solution lies in realizing the relation between call and return arcs. In the example of figure 2a, the definition of dominator analysis has to be modified in order to account for the fact that the arcs C1→E and H→C2 are always executed together. Also, arc C1→E can never be followed by arc H→K2, and arc H→C2 can never be preceded by arc K1→E. Using this fact, a modified dominator analysis routine can conclude that block D is dominated by block B. This in turn makes loop optimizations available on blocks B, C1, C2, and D. For example, instruction LI1 can now be moved out of the loop using loop invariant code motion. On the contrary, instruction LI2 cannot be moved, since it is shared between the loop and blocks K1-K2.

Every analysis and optimization routine in the compiler has to be modified in a similar fashion, in order to take into account call and return arc relations.

3.4. Targeted code specialization

Although the modifications described above increase the optimizer's ability to handle the irregular control flow graphs produced by procedure unification, the resulting optimizer still misses many opportunities that were available during inlining. The reason is that inlining, apart from eliminating calls, also makes a private copy of the callee available for specialization.

Instead of duplicating whole procedure bodies, PBE relies on a *targeted code specialization* phase. A targeted specialization routine can duplicate only selected parts of the program, without being constrained by the original procedure breakup. Therefore it is much less likely than inlining to cause unnecessary code growth. Such a code specialization routine can also benefit all parts of the program, even the ones that don't involve procedure calls.

The first code specialization routine we tried in the context of PBE is an implementation of *superblock formation* that is aware of call and return arc relations. As described in [7], superblock formation identifies hot paths through the code, and then uses tail duplication in order to eliminate side entrances to these paths. Thus each such path can become a single, straight-line "superblock". In our simple example, superblock formation will choose to form a superblock out of the hot path B→C1→E→F→H→C2→D, resulting to the code in Figure 2b.

One can easily see that all optimization opportunities identified in Section 2.2 also exist in the code of Figure 2b. Notice however that the "cold" block G has not been duplicated. Since it is well known that only a small portion of each procedure's code is hot, this difference can lead to significantly less code growth in comparison to inlining.

Although superblock formation is a valuable technique in the context of PBE, a more sophisticated code specialization routine is needed. One of the drawbacks of superblock formation in the context of code specialization is that it can only duplicate a single path through a hot code segment. In many cases making several paths simultaneously available for specialization may be desired. The fact that superblock formation makes its choices relying only on execution weight data is also a drawback. A more sophisticated code duplication method should be able to rule out cases where code duplication does not generate specialization opportunities. Such a method should also be able to duplicate selected portions of the control-flow graph, even portions that do not constitute straight-line paths. We are in the process of developing such a code specialization routine for PBE.

4. Preliminary experimental results

In order to perform a preliminary evaluation of the PBE concept, we used a simple experimental setup. In it, we used the IMPACT compiler [8], in conjunction with procedure unification and superblock formation routines[1] implemented in the Liberty compiler [9]. Since region formation was not available, this system could only handle small benchmarks due to compile time limitations.

Figure 3 compares the results of PBE with those of inlining on three benchmarks: 129.compress from the SPEC'95 benchmark suite, and the UNIX utilities grep and yacc.

Results for inlining were obtained using IMPACT's default inlining routines. For PBE each benchmark was compiled in IMPACT up to a low-level intermediate representation (Lcode). This involved some high-level optimization, but no inlining. The intermediate representation was then saved to a file and passed on to the Liberty compiler for procedure unification and superblock formation. The resulting intermediate representation was fed back to IMPACT for low-level optimization and scheduling.

The first column of the graph in Figure 3 shows the code size of the executables resulting from PBE as a percentage of the code size of the corresponding executables obtained using inlining. The second column does the same for dynamic cycle count. Dynamic cycle counts were obtained by scheduling the programs using IMPACT's scheduling routines for a hypothetical 8-issue EPIC machine, and estimating their cycle counts using profile data, without taking into account cache misses or branch prediction behavior.

As we can see, PBE on average reduces the cycles executed by 16.8%, while producing executables that are 5.7% smaller. The improvements in runtime performance are in fact understated, since our preliminary experiment does not account for the improvement in instruction cache behavior resulting from less code growth. Implementing a better code specialization routine is likely to reduce code growth, improve performance, or both.

5. Conclusion

In this paper we discuss how procedure-centric compiler design methodologies limit the potential of EPIC architectures by limiting the optimization scope available. We then discuss

[1]IMPACT has its own superblock formation routine, but that routine was disabled for the purposes of this experiment.

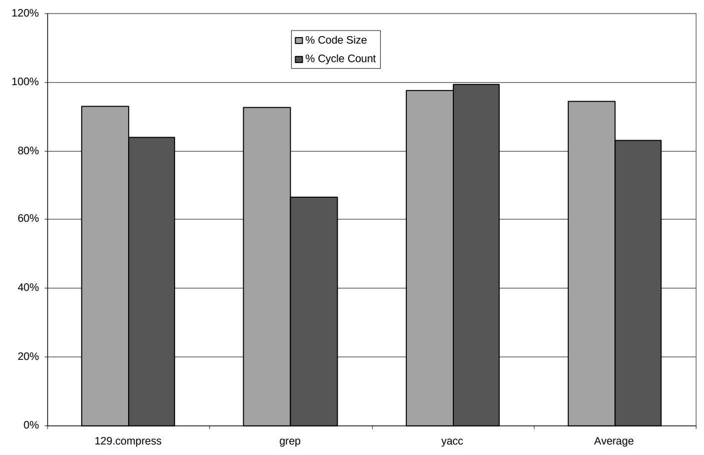

Figure 3: Comparison between inlining and PBE in terms of code growth (a) and static cycle count (b)

how inlining can resolve the issues with optimization scope but only at the cost of code-size. We then propose a new compilation strategy, Procedure Boundary Elimination (PBE) that eliminates procedure boundaries early in the compilation cycle, without code growth. PBE then selects new compilation units via region formation. Since the compiler selects these compilation units, they are selected for optimization potential, and not software engineering concerns as is the case with program procedures. To recover the benefits of code specialization provided by code duplication in inlining, PBE uses a targeted code specialization technique to duplicate code, but only code that will result in performance improvement, instead of entire procedures. In this way PBE achieves all the benefits of inlining without the code size growth and associated performance penalties. After proposing PBE, we discuss the key challenges in building a compiler that uses this technique.

By rethinking the overall strategy that the compiler uses to select compilation units, it should be possible to dramatically improve the performance of EPIC architectures. The preliminary results obtained by implementing only a limited version of PBE are promising, motivating the effort necessary to construct a compiler with full PBE support. We expect that such an implementation of PBE will prove valuable in compilation for EPIC architectures.

References

[1] W. W. Hwu and P. P. Chang, “Inline function expansion for compiling C programs,” in *Proceedings of the ’89 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[2] 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*, vol. 22, pp. 349–370, May 1992.

[3] A. Ayers, R. Schooler, and R. Gottlieb, “Aggressive inlining,” in *Proceedings of the ’97 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 134–145, June 1997.

[4] R. E. Hank, W. W. Hwu, and B. R. Rau, “Region-based compilation: An introduction and motivation,” *International Journal of Parallel Programming*, vol. 25, pp. 113–146, April 1997.

[5] T. Way, B. Breech, and L. Pollock, “Region formation analysis with demand-driven inlining for region-based optimization,” in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 24–33, October 2000.

[6] T. Way and L. L. Pollock, “A region-based partial inlining algorithm for an ILP optimizing compiler,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’02)*, June 2002.

[7] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The superblock: An effective structure for VLIW and superscalar compilation,” tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

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

[9] “The Liberty Project Website.” http://liberty.princeton.edu/.