Self-Evaluating Compilation Applied to Loop Unrolling

Nicholas Nethercote, Doug Burger, and Kathryn S. McKinley

The University of Texas at Austin

Abstract. Well-engineered compilers use a carefully selected set of optimizations, heuristic optimization policies, and a phase ordering to produce good machine code. Designing a compiler with one heuristic per optimization that works well with other optimization phases is a challenging task. Although compiler designers evaluate the optimization heuristics and phase ordering before deployment, compilers typically do not statically evaluate nor refine the quality of their optimization decisions during a specific compilation.

This paper identifies a class of optimizations for which the compiler can evaluate the effectiveness of its heuristics and phase interactions statically, and when necessary re-run optimization phases, using information from the evaluation phase to guide its heuristics. We call this approach self-evaluating compilation (SEC). This model avoids some of the difficulties of predicting phase interactions, and perfecting any one heuristic. The SEC model was motivated by loop unrolling and other optimizations for the TRIPS architecture. TRIPS has a limit on instructions that the compiler can place in an atomic execution unit (a TRIPS block), yet each block has a fixed minimum cost. The goal of loop unrolling (and other optimizations) is to produce as full a block as possible without exceeding the block size, since an unnecessary block with a small number of instruction degrades performance. Because unrolling enables downstream optimizations, it needs to occur well before code generation, but this position makes it impossible to predict the final number of instructions. However, eventually the compiler generates code and can, on a per-loop basis, determine if it unrolled too much or too little or just right. If need be, SEC unrolling then goes back and adjusts the unroll amount accordingly and reruns subsequent optimization phases. We demonstrate a prototype SEC unrolling implementation that automatically matches the best hand unrolled version for a set of microbenchmarks on the TRIPS architectural simulator.

Although motivated by TRIPS compilation challenges, SEC is broadly applicable to helping solve compilation phase ordering and heuristic design for resource constraints such as register and code size limitations which can be measured statically and occur when compiling for embedded, VLIW, and partitioned hardware.

1 Introduction

This paper introduces a self-evaluating compilation (SEC) model in which the compiler adjusts its heuristic policies based on static self-evaluation.

1.1 Compiler Phase Ordering and Heuristic Design

Most compilers include numerous optimization phases. Because finding the optimal code transformation is often NP-complete, most optimizations use heuristic policies. Compiler writers typically tune individual heuristic policies experimentally, based on benchmark behaviors and optimization interactions with previous and subsequent phases. Since phases interact in complex ways, heuristics are not necessarily robust to changes made to another phase or due to phase reordering (e.g., when users specify optimization flags other than the default). To improve the design of individual heuristics, some researchers have turned to machine learning to tune transformation policies [1–3]. To solve the phase ordering problem, compilers have typically relied on a separation of concerns. They typically postpone the handling of resource constraints until the end of compilation, for instance, assuming infinite registers for most of compilation and performing register allocation near the end. Most compilers never evaluate during a specific compilation the quality of their upstream predictions. As more resources become constrained, this separation of concerns degrades the compiler's ability to produce high quality code due to the increasing difficulty of predicting how early decisions influence resource constraints.

Increasing hardware complexity is making this problem harder. For instance, shrinking technology increases clock-speed but exposes wire delays, causing less and less of the chip to be reachable in a single cycle [4]. To address this problem, architects are increasingly partitioning resources such as register banks, caches, ALUs (e.g., partitioned VLIW [5–7] and EDGE architectures [8, 9]), and the entire chip (e.g., chip multiprocessors). Partitioning exposes on-chip latencies and resource constraints to the compiler and thus exacerbates the phase ordering problem and makes the separation of concerns solution with no subsequent static evaluation less appealing.

The example that motivated the SEC approach is creating blocks full of useful instructions for the block atomic execution model in EDGE architectures. Specifically, the TRIPS prototype EDGE architecture has a maximum block size of 128 instructions which the architecture maps at runtime on to a grid of 16 processors which hold 8 instructions per block [8, 10]. This mapping has a fixed overhead. To amortize this overhead and maximize performance, the compiler tries to fill each block with useful instructions while minimizing the total number of blocks. For example, loop unrolling is one method the compiler uses to fill blocks. The compiler performs unrolling early to enable downstream optimizations such as redundant code elimination. However, if it unrolls too much, the resulting code has unnecessary blocks, and if it unrolls too little, each block is less efficient than it could be. Predicting the exact number of instructions in a block early in the compilation cycle requires modeling the effect of all down stream optimizations for the specific loop, and is thus virtually impossible.

A simpler, more familiar example is loop unrolling for conventional architectures. Unrolling enables redundant code elimination and better scheduling, but too much unrolling can degrade instruction cache locality and cause register spilling [11]. Spilling is almost always undesirable, since it increases (1) the number of instructions (loads and stores), (2) latency (two to three cycles in modern caches compared to single cycle register access), and (3) energy consumption due to the cache access. To gain the benefits of unrolling, compilers typically perform it well before register allocation and thus it is difficult for the unroller to predict how its decisions will affect register spilling.

1.2 Improving Optimization Quality

Previous solutions to solving the phase interaction problem include Dean and Chambers who explored static evaluation for inlining on other optimization phases [12] and Brasier et al. who iterated register allocation and instruction scheduling to resolve their tensions [13]. Both of these approaches require all participating phases to annotate and encode the results of their decisions. Thus, each phase must be appropriately engineered for all interacting optimizations, and must be cognizant of their phase ordering and influence on other optimization passes. The SEC model both simplifies and generalizes over these approaches.

Instead of trying to model and perfect all the heuristics and phase interactions, SEC focuses on improving a single optimization heuristic, statically evaluates its decisions after performing other interacting phases, and if necessary, adjusts the heuristic for the particular code fragment and re-optimizes. More formally, given a sequence of ordered compilation phases $\{P_1, P_2, \ldots, P_n\}$, P_i records its optimization decisions. After some later phase P_k, $k > i$, a static evaluation phase P_{E_i} measures the effectiveness of P_i. If P_{E_i} decides P_i's decisions were poor, the compiler uses a checkpoint/rollback mechanism to re-run P_i and the subsequent phases, feeding back information from P_{E_i}'s static evaluation to help P_i adjust its heuristic and do a better job. To minimize the additional compile time, the compiler loops back at most once and/or performs P_i and subsequent phases on just the affected code fragment, e.g., a loop or procedure. For repeated compilation of the same code, using a repository would eliminate repeated iterative compilation [14], or could explore many phase interactions incrementally, thus keeping the per-compile cost overhead down.

For example, SEC unrolling (P_i) check points the intermediate representation, and records how many times it unrolls each loop. After other optimizations, the static evaluation phase (P_{E_i}) counts the instructions in a block. Assume the best unroll amount is n, if the compiler unrolls too little $n-1$, P_{E_i} will go back and unrolling more, if it unrolls too much by $n+2$ P_{E_i} will go back and unroll less. To go back, the compiler discards the code fragments in question, reloads the checkpoint, and invokes the loop unroller again, but adjusts its heuristic using the feedback from P_{E_i} to choose a better unroll factor, and then invokes subsequent phases.

Not all transformations are suited to SEC. The key requirement is that the phase P_i can simply and statically record its decisions, and that P_{E_i} can statically evaluate the resulting code quality. In unrolling, P_i records the number of iterations it unrolls, and the code generator (or a separate P_{E_i} phase) counts the instructions in a block. Suitable SEC transformations include any that increase or change usage of limited resources (such as inlining, unrolling, register allocation, and scheduling), and have simple evaluation functions (such as instruction count, registers, loads/stores, or code size).

This paper makes the following contributions.

- Section 3 presents the self-evaluating compilation (SEC) model that generalizes previous self-evaluating approaches. SEC mitigates the problem of selecting the perfect heuristic and phase ordering. It is suitable for both ahead-of-time and just-in-time compilation and can be implemented in various ways with different simplicity/efficiency trade-offs.
- Sections 4 and 5 evaluate SEC by showing that it improves the effectiveness of loop unrolling for the TRIPS processor which is particularly sensitive to choosing the right unroll factor.
- Section 6 lists additional compiler transformations suited to SEC, and a useful variation called *cloning* SEC.

Although SEC was motivated by our particular compilation problem on TRIPS, it is applicable in many more settings. For instance, embedded programs also have strict instruction space requirements and limited register files. Compilers for embedded processors could more easily assess the costs and benefits of code expanding (e.g., inlining, unrolling) and code size reduction (e.g., procedure abstraction, inlining) transformations with the SEC model.

2 Related Work

This section compares SEC with related work on selecting good optimization heuristics, phase ordering, and other compiler feedback loops.

2.1 Designing Optimization Heuristics

The difficulty of designing good heuristics for individual optimizations is witnessed by a diversity of advanced approaches [3, 15, 1, 2, 16] that automate this process. For example, Stephenson et al. [1, 2] use genetic algorithms to derive hyperblock formation, register allocation, and prefetching; and supervised learning for unrolling. Cavazos and Moss [3] use supervised learning to decide whether to schedule blocks in a Java JIT. SEC is complementary to learning heuristics that are likely dependent upon phase ordering.

2.2 Phase Interactions

The most closely related work [13, 12] performs static self-evaluation, but is not as general or as simple as SEC. Brasier et al. use a static feedback loop to reduce the antagonism between instruction scheduling and register allocation in a system called CRAIG [13]. It first performs instruction scheduling followed by register allocation; if it spills too much, it starts over and does register allocation before performing incremental instruction scheduling, moving towards late assignment. CRAIG thus reorders phases and refines its scheduling heuristic based on spill feedback. SEC generalizes beyond these two closely related phases by communicating more information across more phases.

Dean and Chambers use inlining trials to avoid designing a heuristic that models the exposed optimization opportunities for subsequent phases [12]. After inlining, each subsequent phase must carefully track how inlining influenced its decisions. If inlining does not enable optimizations that reduce the resulting code size (a static measure), the compiler reverses the inlining decision, recompiles the caller, and records the decision to prevent future inlining of this method and similar ones. This approach is limited because it requires changes to all subsequent phases for each optimization it wishes to evaluate. Section 6 describes how SEC inlining eliminates the need to change all intervening phases.

2.3 Iterative Compilation

Iterative compilation seeks to evaluate the extent of the phase ordering problem and to improve over default orderings. Iterative compilation turns on or off and reorders phases and empirically evaluates hundreds or thousands of compile-execute sequences to find the configuration that produces the fastest or smallest code for a particular program. Results show that default compiler phase orderings and settings are far from optimal [17–20]. Because the configuration space is huge, researchers use search techniques such as genetic algorithms and simulated annealing to reduce the number of compile-execute cycles. In comparison, SEC chooses one fixed order, but incurs substantially smaller compile time overheads. Iterative compilation also has the disadvantage that if the compiler changes (e.g., adds a new optimization or changes a heuristic), one should perform the trials again. SEC is more robust to compiler changes.

2.4 Feedback-Directed Optimization

Smith defines feedback-directed optimizations (FDO) [21–24, 14] as a family of techniques that alter a program's execution based on tendencies observed in its present or past runs. FDO is orthogonal to SEC since it typically uses dynamic edge, path, or method profiling to select which code to optimize, and to guide heuristics. Arnold et al. add a repository to combine online and offline (previously profiled) optimization plans that incorporate the costs and benefits of online optimization [14]. SEC instead feeds back static evaluations, e.g., code size or register usage, and then adjusts heuristics to avoid phase ordering problems, all without ever executing the program. SEC and FDO are thus complementary.

3 Self-Evaluating Compilation

This section describes SEC in three parts. First, it explains SEC's most general form for ahead-of-time compilation. Second, it discusses three specific instances of SEC, with different simplicity/efficiency trade-offs. Finally, it describes how SEC can complement just-in-time compilation.

3.1 General SEC

SEC involves the following compiler phases.

```
run $pre(P_{CP})$ on all fragments (e.g., loop, module, etc.)          1
for each fragment $F$                                                  2
    $F_{info}$ := (empty);  $N := 0$                                   3
    run $P_{CP}$ on $F$                                                4
    while (true)                                                       5
        run $between(P_{CP}, P_i)$ on $F$                              6
        run $P_i$ on F, using $F_{info}$ if non-empty                  7
        run $between(P_i, P_{E_i})$ on $F$                             8
        if $N <$ the loop limit                                        9
            run $P_{E_i}$ on $F$, and record $F_{info}$                10
        run $between(P_{E_i}, P_{LB})$ on $F$                          11
        run $P_{LB}$: if ($N =$ loop limit or $F$ evaluated ok)        12
            exit inner loop                                            13
        run $P_{RB}$ on $F$                                            14
        $N := N + 1$                                                   15
run $post(P_{LB})$ on all fragments                                    16
```

Fig. 1. Pseudocode of general SEC. *pre*(P) gives the phases that precede P; *between*(P, Q) gives the phases between P and Q; *post*(Q) gives the phases that come after Q.

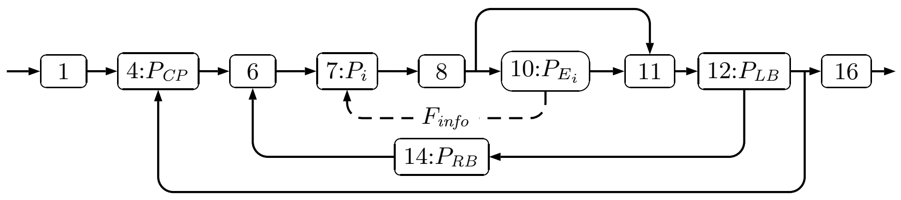

Fig. 2. Control flow of general SEC. Solid lines represent control flow, dashed lines represent data flow. The number labels correspond to pseudocode lines in Figure 1.

P_{CP}:	Checkpoints code fragments (e.g. saves them to memory or file).
P_i:	Transforms/optimizes the code using a heuristic.
P_{E_i}:	Evaluates the effectiveness of the P_i transformation.
P_{LB}:	Determines if the current code fragment is acceptable or should be recompiled with a modified heuristic, based on the results of P_{E_i} and the number of times P_i has executed.
P_{RB}:	Rolls back to a checkpointed code fragment.

Figure 1 gives the pseudocode for the most general form of SEC. Figure 2 shows its control flow and data flow in a diagram. SEC involves two main loops. The inner loop is the heart of SEC. It processes one code fragment at a time, where a fragment could be a loop, a procedure, or even a whole module. It performs the transformation phase P_i and subsequent phases, then runs P_{E_i} to evaluate P_i's decisions. It uses P_{RB} to roll back the fragment to the version checkpointed by P_{CP} if necessary, and repeats until the fragment F is deemed "good enough". The outer loop iterates through the code fragments one at a time.

P_i and P_{E_i} interact in three key ways. First, *P_i must indicate to P_{E_i} what decisions it made.* P_i does this by annotating the code. In our example, P_i identifies the unrolled loop and the unroll factor to P_{E_i}. Second, *P_{E_i} must be able to meaningfully evaluate P_i's decisions.* P_{E_i} must include a static measurement that evaluates code quality. A simple example is "this unrolled loop has too many spills." The performance improvements obtained with SEC depend heavily on the accuracy of P_{E_i}'s evaluations. Third, *P_{E_i} should help P_i improve its decisions on any rejected fragment.* In our example, P_{E_i} could indicate that the chosen unroll factor was too high (e.g., more blocks than necessary and at least one under-full block) or too low (e.g., one under-full with room for more unrolled iterations), or just right. If needed, the unrolling heuristic then can change the unroll factor accordingly. Figure 2 depicts this information flow with a dashed line.

Thus far we have discussed using one P_i/P_{E_i} pair, but the compiler can use more. For example, one could employ a single large inner loop: perform all the P_i transformations on a fragment, then later do all the P_{E_i} evaluations, and if any transformations were unsatisfactory, loop back and redo them all. This requires only checkpointing once, but all P_i phases would be re-run even if only a single P_{E_i} evaluation failed. Alternatively, one could employ more checkpointing and have multiple inner loops, which might overlap or be entirely separate. The best configuration would depend on how the different P_i phases interact and the increase in compile time the system is willing to tolerate.

3.2 Specific Instances of SEC

We can partially or fully instantiate general SEC by specifying some or all of its parameters: the size of each fragment F, the loop limit N, the workings of the phases P_i, P_{E_i}, P_{CP}, P_{RB}, P_{LB}, and the phases present in each of $\mathit{pre}(P_{CP})$, $\mathit{between}(P_{CP}, P_i)$, $\mathit{between}(P_i, P_{E_i})$, $\mathit{between}(P_{E_i}, P_{LB})$ and $\mathit{post}(P_{LB})$. The following paragraphs describe three such general instances, and Section 4 presents an SEC optimization: unrolling for TRIPS.

A complex but efficient instance. Code fragments are procedures, and the inner loop executes at most twice. P_{CP} and P_{RB} are simple: P_{CP} saves a copy of a procedure in memory, and P_{RB} discards the changed code and reverts to the saved code. The phase sequences $\mathit{between}(P_{CP}, P_i)$ and $\mathit{between}(P_{E_i}, P_{LB})$ are empty. We do not specify the other parameters—P_i, P_{E_i}, $\mathit{pre}(P_{CP})$, $\mathit{between}(P_i, P_{E_i})$ and $\mathit{post}(P_{LB})$—so this instance still has some generality.

This instance is efficient—it minimizes the number of phases that are re-run, and the small fragment size reduces phase re-running times and the size of the checkpoint. However, it requires some structural support in the compiler, e.g., the ability to run multiple phases in succession on a single procedure. Procedures are a good general choice for fragments as they are not too big, but are still standalone units. Smaller units are possible, but it would require more work to join up a re-compiled fragment with other sub-procedure fragments. The efficiency of this instance depends on how often P_i makes bad decisions; if it makes no bad decisions, no phase re-running will occur at all.

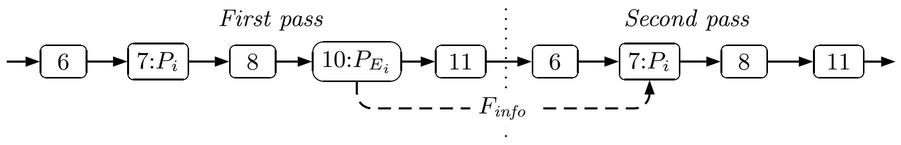

Fig. 3. Control flow of an easy-to-implement instance of SEC. Solid lines represent control flow, dashed lines represent data flow. The number labels correspond to pseudocode lines in Figure 1.

A simpler instance. A variant of the first instance involves changing the size of the code fragments to an entire module, which effectively removes the outer loop. This structure is less efficient—checkpointing and possibly re-running phases for the entire module—but simpler to implement. Note that although the fragment size is a whole module, the heuristics are adjusted at a finer level. For example, with loop unrolling P_{E_i} will include information about every unrolled loop in F_{info}. As a result, the fragment size does not affect accuracy, only the amount of checkpointing and phase re-running performed.

An easy-to-implement instance. At the other end of the complexity/efficiency spectrum is the instance shown in Figure 3. The compiler runs to completion twice. In the first pass P_{E_i} writes F_{info} for the whole module to a file. In the second pass P_i reads F_{info} from this file and adjusts its decisions accordingly; P_{E_i} does not need to be re-executed. Each code fragment is an entire module. P_{CP} and P_{RB} are no-ops; since the compiler runs twice, the original source code serves as the checkpoint. Both $pre(P_{CP})$ and $post(P_{LB})$ are empty.

This instance is an excellent way to trial SEC in an existing compiler, since the only change needed is support for writing and reading the data file; the inner loop can be implemented by a wrapper script. (We implemented SEC exactly this way for our evaluation in Section 5; Section 4.3 has more details.) And even though it sacrifices efficiency for simplicity, it only doubles compilation time.

4 An SEC Example: Loop Unrolling for TRIPS

In this section we review loop unrolling, describe the TRIPS architecture and the unique challenges it poses to the loop unroller, and explain how we use SEC to guide loop unrolling in the TRIPS compiler.

4.1 Loop Unrolling

Loop unrolling is a common transformation which duplicates a loop's body one or more times. The *unroll factor* is the number of copies of the loop body in the final unrolled loop; an unroll factor of one means no unrolling. The simplest case is when the loop trip count is known statically and the unroll factor divides it evenly; the unrolled loop on the right has an unroll factor of three.

```
for (i = 0; i < 120; i++) {          for (i = 0; i < 118; i += 3) {
   b(i);                        =>      b(i);  b(i+1);  b(i+2);
}                                    }
```

We restrict this discussion to loops with sufficiently large trip counts, but the same framework flattens loops with small trip counts. If the loop trip count is known statically but the unroll factor does not divide it evenly, or the loop trip count is statically unknown but invariant, a "clean-up" loop performs the final few iterations. If the trip count is known statically, the clean-up loop can be flattened.

```
for (i = 0; i < n; i++) {          for (i = 0; i < n-2;  i += 3) {
    b(i);                     =>       b(i);  b(i+1);  b(i+2);
}                                  }
                                   for ( ; i < n; i++) {
                                       b(i);
                                   }
```

Loop unrolling can improve program performance on traditional architectures in two ways. First, the unrolled loop requires fewer instructions, because there are fewer loop tests and backward branches, and in some forms, fewer updates of the index variable. Second, it enables other optimizations. For example, if the body loads or stores the same memory location on distinct adjacent iterations, scalar replacement can replace some memory accesses with register accesses. Also, loop unrolling exposes instruction level parallelism (ILP) allowing better instruction scheduling.

Loop unrolling may degrade performance if loops are unrolled too much. For example, it can increase register pressure if scalar replacement uses too many registers and causes spilling. Another potential problem is that the increased code size can degrade the performance of the instruction cache.

4.2 TRIPS

TRIPS is a prototype implementation of a new class of microprocessor architectures called EDGE (Explicit Data Graph Execution) designed to provide high performance and low power consumption in the face of technology trends such as increasing clock speed and increasing wire delays [8, 25, 10]. The prototype design is complete and working chips should be operational in 2006.

Unlike traditional architectures that operate at the granularity of a single instruction, EDGE ISAs are based on blocks of instructions. EDGE architectures implement serial, block-atomic execution, mapping each block on to the ALU grid (as depicted in Figure 4), executing it atomically, committing it, and fetching the next block. Execution within blocks is purely dataflow, so that each instruction forwards its results directly to its consumers in that block without going through a shared register file (registers are only used for inter-block value passing). Each block is a hyperblock—a single-entry, multiple-exit set of predicated basic blocks [26, 27]—with some additional constraints [8].

In the TRIPS prototype, a key constraint is that each block is fixed-size and holds at most 128 instructions. Each ALU in the 4×4 ALU grid has up to 8 instructions in each block mapped onto it ($4 \times 4 \times 8 = 128$). Up to 16 instructions can execute per cycle, one per ALU. Up to 8 blocks can be in-flight at once (7 of them speculatively), resulting in a 1,024-wide instruction window.

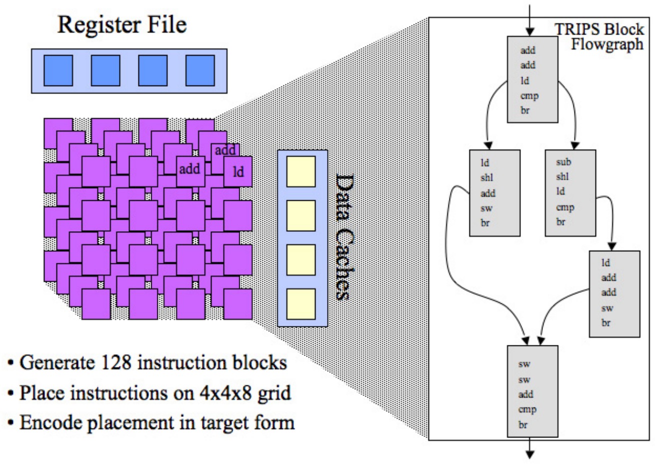

Fig. 4. The TRIPS prototype compilation target.

The biggest challenge for the TRIPS compiler [10] is to create blocks full of useful instructions; ideally it will produce high quality blocks with close to 128 instructions, maximizing ILP and minimizing the fixed per-block execution overheads. To achieve this goal, the compiler's main tools are the use of predication to include multiple basic blocks in each TRIPS block, inlining, and loop unrolling. The TRIPS compiler also uses SEC block combining.

4.3 Challenges for Loop Unrolling on TRIPS

To maximize TRIPS performance, the loop unroller would ideally produce unrolled loops containing exactly, or slightly less than, 128 instructions. This requirement presents quite a challenge for upstream optimizations such as unrolling. Consider a loop with a known trip count of 120, in which each loop body has 31 instructions, and the loop test and exit are 4 instructions. If the unroll factor is 3, the block size is $31 \times 3 + 4 = 97$ instructions, and the loop will execute in 40 blocks.

In comparison, an unroll factor of 4 yields a block size of $31 \times 4 + 4 = 128$ instructions, and the loop will execute in only 30 blocks. However, if the size of the loop body is 32 instructions, an unroll factor of four produces a loop size of 132 instructions which requires two TRIPS blocks, and 60 block executions. An unroll factor of 2 produces this same result.

This example demonstrates that accurate instruction counts are vital for loop unrolling on TRIPS; in some cases, even underestimating the size of a loop body by one instruction can harm code quality. This example assumes that all duplicated loop bodies have the same size, but downstream optimizations like common subexpression elimination and test elision will change the resulting code size further complicating the unrollers job. The TRIPS compiler performs unrolling early in the optimization sequence because many optimizations, such as scalar replacement, can further improve code quality after unrolling. Therefore, accurate instruction counts are not available to the unroller and become available only near the end of compilation. We use SEC to solve this problem.

4.4 SEC Heuristic Adjustment

We currently use the simple two-pass SEC instance described in Section 3.2. In the first pass, the compiler performs the phases $pre(P_i)$ (represented by box 6 in Figure 3), which include parsing, inlining, and conversion to a static-single assignment, control flow graph intermediate (IR) form. The loop unroller (P_i) then executes for the first time. For each candidate loop, it estimates the number of TRIPS instructions in the loop body (S_B), and the loop and exit tests (S_E) by examining each IR instruction in the loop. If $S_B + S_E > 128$, no unrolling takes place, otherwise it selects an unroll factor of $U_1 = (128 - S_E)/S_B$. Loops that are larger than half a block are therefore never unrolled; we have experimented with trying to unroll to fit three loop bodies in two blocks, but with little success. One exception is that the unroller flattens loops with a known loop bound up to 512 instructions. Because flattened loops have no back edges, subsequent phases can easily merge them with surrounding blocks.

After unrolling, P_i marks the unrolled loops with the estimated sizes. The compiler then performs the phases in $between(P_i, P_{E_i})$ (box 8 in Figure 3), which in the TRIPS compiler include many optimizations (scalar replacement, constant propagation, global value numbering, etc.), code generation, and hyperblock formation. P_{E_i} then measures the actual sizes of the unrolled loops—accurate measurements are now possible—and writes its measurements (F_{info}) to the data file. The final phases in $post(P_{E_i})$ then run; they include splitting of too-large blocks, register allocation, and instruction scheduling.

The second pass is like the first except P_i does not use IR-based loop size estimation. Instead, for each unrolled loop it reads from the data file (F_{info}) the measured size of the entire unrolled loop (S_L) and the loop test and exit (S_{E2}). It then estimates the size of each loop body as $S_{B2} = (S_L - S_{E2})/U_1$. This estimate is imperfect because the iterations in an unrolled loop are not always exactly the same size, but it improves over the IR-based estimate (Section 5.1 quantifies the difference). With this more accurate loop size estimate, it can now compute the new unroll factor, $U_2 = (128 - S_{E2})/S_{B2}$, and compilation continues to the end.

This application of SEC results in fuller TRIPS blocks, reducing the block execution count and speeding up programs, as the next section shows.

5 Evaluation

This section evaluates SEC's effectiveness in improving loop unrolling for TRIPS. Because TRIPS hardware is not yet available, we use a simulator for our experiments. It is cycle-accurate and slow, so we use only microbenchmarks for our evaluation. The suite consists of 14 microbenchmarks containing key inner loops extracted from SPEC2000, five kernels from an MIT Lincoln Laboratory radar benchmark (doppler_GMTI, fft2_GMTI, fft4_GMTI, transpose_GMTI, forward_GMTI), a vector add kernel (vadd), a ten-by-ten matrix multiply (matrix_1), and a discrete cosine transform (dct8x8). Because these benchmarks are all dominated by loops, the unrolling benefits should be large.

	Unrolling Policies							
	Block Count Reduction				Cycle Count Reduction			
	none	by 3%	Est%	SEC%	none	by 3%	Est%	SEC%
art_2	22061	66.3	66.3	84.9	504565	61.6	61.6	79.1
vadd	54334	84.0	84.1	84.1	439449	71.9	73.3	73.3
transpose_GMTI	78349	83.8	83.8	83.8	1027366	75.4	75.5	75.5
matrix_1	24665	81.5	81.5	81.5	383814	68.7	68.7	68.7
art_3	30051	49.9	76.8	80.9	443259	47.8	70.0	68.9
art_1	16651	64.3	66.2	78.9	233755	50.5	56.7	67.4
twolf_1	38631	53.7	65.3	76.6	689344	57.7	58.4	62.7
twolf_3	14051	49.8	49.7	66.3	673539	2.3	2.8	6.3
gzip_1	2395	32.9	54.7	57.3	24664	-10.1	-5.8	-2.3
bzip2_2	32911	44.7	51.7	52.5	426155	31.4	38.8	41.0
bzip2_1	15682	0.2	0.2	49.6	410409	30.9	31.2	39.5
doppler_GMTI	11190	43.8	12.5	37.5	396943	21.0	15.7	24.4
equake_1	16405	62.2	37.3	37.3	331378	55.1	43.9	43.9
gzip_2	8986	40.6	30.2	30.2	129110	48.5	31.7	31.7
forward_GMTI	11825	-5.1	13.5	13.5	392571	11.0	14.7	14.7
ammp_2	30951	5.5	16.5	11.0	910693	32.1	36.0	34.7
ammp_1	60751	0.0	8.9	8.9	1891762	0.0	-8.1	-8.1
dct8x8	3046	9.2	4.2	4.2	61756	11.8	41.2	41.2
parser_1	7051	-33.0	0.0	0.0	225255	11.6	6.1	0.0
bzip2_3	15531	0.0	-48.3	0.0	400906	14.0	-1.6	0.0
fft4_GMTI	9745	-9.2	-8.2	-8.2	142233	-8.8	-10.6	-10.6
fft2_GMTI	8378	-23.3	-9.5	-9.5	245022	-10.9	-2.7	-2.7
arith. mean		31.9	33.5	41.9		30.6	31.7	34.1

Table 1. Block and cycle results for unrolling. Column 1 gives the benchmark name. Column 2 (none) gives the number of blocks executed with no unrolling. Columns 3–5 give the percentage reduction in the number of blocks executed vs. no unrolling: column 3 (by 3%) is default unrolling, column 4 (Est%) is IR size estimation only, column 5 (SEC%) is SEC. Columns 6–9 give the corresponding cycle results.

5.1 Goals

SEC unrolling has three goals: 1) to improve the size estimates and thus produce fuller blocks; 2) to execute fewer blocks at runtime because each block is fuller; and 3) to ultimately speed up the program by reducing the time required to map blocks onto the ALU grid, and by exposing more ILP.

5.2 Results

SEC works well in improving the unroller's estimates of loop sizes, the first goal. The microbenchmark suite has 33 candidate loops. On the first pass, using the IR-based loop size estimator, the average relative error of this estimate was 61%. On the second pass, using the back end measurements of the resulting loop sizes, the average relative error was 6%. Most of the estimates by the second pass were within 1 or 2%; the worst result was a 24% underestimate. Poor final estimates correlated with the worst initial estimates, since the unroll factor chosen in the second pass was quite different from the first pass, and thus there was more room for error. Performing this cycle a third time attains small improvements, but is not worth the extra compilation time.

Table 1 quantifies how well SEC unrolling achieves the second and third goals. We perform the following four experiments.

- No unrolling (none).
- Default unrolling (by 3): unroll by three for unknown loop bounds; for known bounds, flatten loops if the flattened size is less than 200 statements; otherwise choose an unroll factor based on an IR estimate.
- IR code estimation (Est): the compiler estimates the number of TRIPS instructions from the IR.
- SEC corrected estimation (SEC): uses the two-pass compiler structure and phases described above.

On average, SEC decreases the average number of blocks executed by 10% over the default unroller, and 8% over Est. It attains this result by substantial improvements over default unrolling on art_2, art_3, twolf_1, twolf_3, and bzip2_1, and avoiding the substantial degradations of Est and default on parser_1, bzip2_3, and fft2_GMTI. For art_2 and art_3 the improvement comes from SEC using higher unroll factors (6 and 8 instead of 3) for loops with small bodies; for the others the improvement comes from choosing lower unroll factors (e.g. 2 instead of 3, 3 instead of 6) for loops with larger bodies, so that the entire unrolled loop fits within one block. Est is only marginally better than the default non-TRIPS-specific policy, which shows that the TRIPS-specific unrolling policy is of little use without SEC's accurate size estimates.

The cycle count improvements with SEC are smaller than the block count improvements. This is due to architectural complications that are beyond the scope of this paper. However, these results are more encouraging than they seem. Currently the TRIPS compiler does not perform instruction-level hyperblock optimizations, so the code it produces still has some "fat". Experiments with hand-coded microbenchmarks show that cycle count improvements due to loop unrolling increase as code quality goes up. For example, we have seen that better scalar replacement reduces twolf_3's run-time by around 20%.

Taken together, the dynamic block size and block execution results demonstrate that SEC can help bridge the phase ordering problem and provide improved heuristics for loop unrolling.

6 Discussion

This section describes other potential SEC optimizations SEC instances for mitigating its compilation time, and how to integrate SEC in a JIT compiler and with interprocedural optimizations.

6.1 Other Potential SEC Optimizations

This section describes other optimizations that interact with resource usage and are thus amenable to static evaluation with SEC.

- Loop unrolling can also increase register spilling. P_{E_i} could decide to reject any loop in which unrolling causes a register spill.

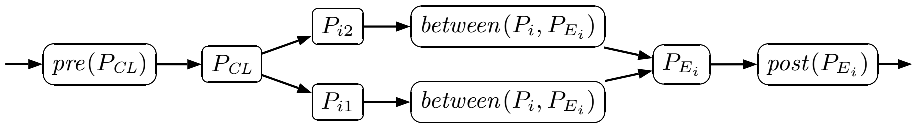
$$\begin{tikzcd}
 & & P_{i2} \arrow[r] & between(P_i, P_{E_i}) \arrow[dr] & & \\
\arrow[r] & pre(P_{CL}) \arrow[r] & P_{CL} \arrow[u] \arrow[d] & & P_{E_i} \arrow[r] & post(P_{E_i}) \arrow[r] & {} \\
 & & P_{i1} \arrow[r] & between(P_i, P_{E_i}) \arrow[ur] & &
\end{tikzcd}$$

Fig. 5. Control flow of cloning SEC.

- We are currently integrating SEC into the TRIPS compiler to improve TRIPS block formation. When considering the inclusion of multiple basic blocks for a single TRIPS block, it is difficult to estimate the resulting block size beforehand because of subsequent optimizations (just like unrolling for block size). Instead, we optimistically and incrementally combine blocks, run the optimizations on the the resulting hyperblock, and roll back if it exceeds 128 instructions.

A variant called *cloning SEC* removes the feedback loop but still uses static evaluation. Its control flow is shown in Figure 5. Instead of P_{CP} checkpointing the code, a cloning phase P_{CL} makes one or more temporary clones of a code fragment. P_i then runs in a different way on each clone. After the intervening phases are run on each clone, P_{E_i} then statically compares the clones and chooses the best one, discarding the others. This approach is most useful when P_i can choose between only a small number of possible transformations (e.g. whether to inline a particular procedure call or not). The following list gives some cases where this variation might be applicable.

- If P_i is an inliner, P_{E_i} could evaluate whether inlining increases register spills or bloats the code too much by comparing fragments in which calls were inlined and fragments in which they were not. This approach achieves a similar goal to inlining trials [12] but does not require the phases between P_i and P_{E_i} to track any additional information. This application is interesting for embedded platforms where code size is critical.
- Procedure abstraction—in which the compiler factors out matching code sequences into a procedure [28]—is sometimes used to reduce code size. However code size might increase if the register allocator must spill around the call. Cloning SEC could choose the clone that did not have procedure abstraction applied if it ended up being smaller.
- The TRIPS compiler's back end estimates code size before splitting too-large blocks, as Section 4.3 described. Sometimes its estimates are inaccurate, partly because the compiler performs some optimizations (e.g. peephole optimizations) after block splitting, and partly because the estimation is conservative in various ways. Cloning SEC could select an unsplit block if it ends up fitting within 128 instructions.
- Vectorizing compilers targeting SIMD instruction sets will often unroll a loop by 2 or 4 with the goal of converting groups of scalar operations into single SIMD instructions. Scalar code in the loop body can prevent this transformation from working, in which case no unrolling is probably preferable, but

it is difficult to predict at loop unrolling time. Cloning SEC could be used to discard any unrolled loops that failed to be vectorized.

The common theme to these uses is that P_i transforms the code in a way that can result in better (faster and/or smaller) code, but puts stress on a limited machine resource such as registers or TRIPS block sizes. P_{E_i} then evaluates whether the increased resource stress is just right or too much or if more stress could be tolerated. However, SEC must be applied judiciously to select optimizations due its increased compile time that is proportional to the number of SEC optimizations.

6.2 Just-In-Time and Interprocedural Compilation

SEC can be used as-is in a just-in-time (JIT) compiler. However, many JIT compilers use staged dynamic optimization. This mechanism offers an opportunity to improve the efficiency of self-evaluation by eliminating the usual re-running of phases. Instead, the JIT compiler can piggyback the inner loop phases P_{LB} and P_{RB} onto its existing recompilation loop. This formulation would also complement Arnold et al.'s repository for combining on-line and off-line profiling in a JIT by providing more accurate benefit measurements [**?**].

SEC would operate on the JIT's existing recompilation unit (e.g., methods) along with its existing P_{CP} and P_{RB} phases. When P_{E_i} evaluates P_i's decisions, rather than immediately rejecting substandard code, it records F_{info} (e.g., code size, register spills, etc.) and executes this initial version of the code. If a fragment is hot and worth recompiling, P_{RB} is invoked on it as usual, and P_i can use F_{info} to improve its heuristics during recompilation. This structure eliminates the additional compile-time cost that SEC incurs, requires no additional checkpointing cost (because it utilizes the JIT compiler's existing checkpointing mechanism), and still gains its benefits for hot methods. The only addition required is the ability to record F_{info} (which is typically compact) for each code fragment.

Another important consideration for SEC is interprocedural analysis. If the fragment size is less than a whole module, any interprocedural analysis must take place during $pre(P_{CP})$ or $post(P_{LB})$; any analysis between P_{CP} and P_{LB} might be invalidated by the per-fragment re-running of phases. This requirement is unlikely to cause problems in practice.

7 Conclusion

We presented a general model of self-evaluating compilation (SEC), in which a compiler adjusts its heuristic policies based on static self-evaluation. Implementing SEC efficiently in an ahead-of-time compiler requires significant effort, but one can easily test if it will be worthwhile with the simple two-pass instance from Section 3.2. JIT compilers with multiple levels of optimization can easily incorporate SEC by piggybacking SEC's evaluation and heuristic tuning on existing recompilation frameworks. These characteristics are desirable in a field that is full of clever but complex ideas that do not make it into production compilers—

as Arch Robison noted [29]: “Compile-time program optimizations are similar to poetry: more are written than actually published in production compilers.”

This paper also identified previous instances of static self-evaluation in the literature, showed how SEC generalizes them, and described a number of additional optimization heuristics and phase orderings which could benefit from this approach. It illustrated effective SEC loop unrolling for the TRIPS architecture in which the compiler corrected its unrolling heuristic to meet the TRIPS block size constraints in a simple two pass SEC instance. Furthermore, simulation results demonstrated that this self-adjusting heuristic improved performance by reducing the number of executed blocks.

References

1. Stephenson, M., Amarasinghe, S., Martin, M.C., O’Reilly, U.M.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of PLDI 2003, San Diego, CA (2003)
2. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: The International Conference on Code Generation and Optimization, San Jose, CA (2005)
3. Cavazos, J., Moss, J.E.B.: Inducing heuristics to decide whether to schedule. In: Proceedings of PLDI 2004, Washington, DC (2004) 183–194
4. Agarwal, V., Hrishikesh, M., Keckler, S.W., Burger, D.: Clock rate versus IPC: The end of the road for conventional microarchitectures. In: Proceedings of the 27th International Symposium on Computer Architecture. (2000) 248–259
5. Kailas, K., Ebcioglu, K., Agrawala, A.K.: CARS: A new code generation framework for clustered ILP processors. In: International Symposium on High-Performance Computer Architecture. (2001) 133–143
6. Kessler, C., Bednarski, A.: Optimal integrated code generation for clustered VLIW architectures. In: Joint Conference on Languages, Compilers and Tools for Embedded Systems. (2002) 102–111
7. Zhong, H., Fan, K., Mahlke, S., Schlansker, M.: A distributed control path architecture for vliw processors. In: International Conference on Parallel Architectures and Compilation Techniques, Washington, DC (2005) 197–206
8. Burger, D., Keckler, S.W., McKinley, K.S., et al.: Scaling to the end of silicon with EDGE architectures. IEEE Computer (2004) 44–55
9. Nagarajan, R., Burger, D., McKinley, K.S., Lin, C., Keckler, S.W., Kushwaha, S.K.: Instruction scheduling for emerging communication-exposed architectures. In: International Conference on Parallel Architecture and Compiler Techniques, Antibes Juan-les-Pins, France (2004) 74–84
10. Smith, A., Burrill, J., Gibson, J., Maher, B., Nethercote, N., Yoder, B., Buger, D., McKinley, K.S.: Compiling for edge architectures. In: The International Conference on Code Generation and Optimization. (2006) To appear.
11. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. In: Proceedings of PLDI 1990, White Plains, NY (1990) 53–65
12. Dean, J., Chambers, C.: Towards better inlining decisions using inlining trials. In: Proceedings of LFP ’94, Orlando, FL (1994) 273–282
13. Brasier, T.S., Sweany, P.H., Carr, S., Beaty, S.J.: CRAIG: A practical framework for combining instruction scheduling and register assignment. In: International Conference on Parallel Architecture and Compiler Techniques, Cyprus (1995)

14. Arnold, M.R., Welc, A., Rajan, V.T.: Improving virtual machine performance using a cross-run profile repository. In: ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications. (2005) 297–311
15. Moss, J.E.B., Utgoff, P.E., Cavazos, J., Precup, D., Stefanovic, D., Brodley, C., Scheeff, D.: Learning to schedule straight-line code. In: Neural Information Processing Systems – Natural and Synthetic, Denver, CO (1997)
16. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M.J., Padua, D., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: Proceedings of PLDI 2003, San Diego, CA (2003) 63–76
17. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: Proceedings of LCTES 2004, Washington, DC (2004)
18. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of LCTES '99, Atlanta (1999) 1–9
19. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Automatic selection of compiler options using non-parametric inferential statistics. In: International Conference on Parallel Architecture and Compiler Techniques, St. Louis, MO (2005) 123–132
20. Ladd, S.R.: Acovea: Using natural selection to investigate software complexities (2003) http://www.coyotegulch.com/products/acovea/.
21. Alpern, B., Attanasio, D., Barton, J.J., et al.: The Jalapeño virtual machine. IBM System Journal **39** (2000)
22. Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.: A survey of adaptive optimization in virtual machines. IEEE Computer **93** (2005)
23. Smith, M.D.: Overcoming the challenges to feedback-directed optimization. In: Proceedings of Dynamo '00, Boston, MA (2000) 1–11
24. Sun MicroSystems: The Sun HotSpot compiler (2005) http://www.sun.com/software/communitysource/hotspot/.
25. Nagarajan, R., Sankaralingam, K., Burger, D., Keckler, S.W.: A design space evaluation of grid processor architectures. In: Proceedings of MICRO34, Austin, TX (2001) 40–53
26. Fisher, J.A.: Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers **C-30** (1981) 478–490
27. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: Proceedings of MICRO25, Portland, OR (1992) 45–54
28. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences **52** (1996) 28–42
29. Robison, A.D.: Impact of economics on compiler optimization. In: Proceedings of the ACM 2001 Java Grande Conference, Palo Alto, CA (2001) 1–10