Journal of Instruction-Level Parallelism 7 (2005) 1-25 Submitted 10/04; published 2/05

Compiler Optimization-Space Exploration

Spyridon Triantafyllis STRIANTA@PRINCETON.EDU
Department of Computer Science
Princeton University
Princeton, NJ 08540 USA
Manish Vachharajani MANISHV@COLORADO.EDU
Department of Electrical and Computer Engineering
University of Colorado at Boulder
Boulder, CO 80309 USA
David I. August AUGUST@PRINCETON.EDU
Department of Computer Science
Princeton University
Princeton, NJ 08540 USA

Abstract

To meet the performance demands of modern architectures, compilers incorporate an ever-increasing number of aggressive code transformations. Since most of these transformations are not universally beneficial, compilers traditionally control their application through *predictive heuristics*, which attempt to judge an optimization's effect on final code quality *a priori*. However, complex target architectures and unpredictable optimization interactions severely limit the accuracy of these judgments, leading to performance degradation because of poor optimization decisions.

This performance loss can be avoided through the *iterative compilation* approach, which advocates exploring many optimization options and selecting the best one *a posteriori*. However, existing iterative compilation systems suffer from excessive compile times and narrow application domains. By overcoming these limitations, Optimization-Space Exploration (OSE) becomes the first iterative compilation technique suitable for general-purpose production compilers. OSE narrows down the space of optimization options explored through limited use of heuristics. A compiler tuning phase further limits the exploration space. At compile time, OSE prunes the remaining optimization configurations in the search space by exploiting feedback from earlier configurations tried. Finally, rather than measuring actual runtimes, OSE compares optimization outcomes through static performance estimation, further enhancing compilation speed. An OSE-enhanced version of Intel's reference compiler for the Itanium architecture yields a performance improvement of more than 20% for some SPEC benchmarks.

1. Introduction

An aggressively optimizing compiler is essential for achieving good performance on modern processors. Non-uniform resources, explicit parallelism, multilevel memory hierarchies, speculation support, and other advanced performance features of modern processors can only be exploited if the compiler can effectively target them. This dependence on compiler quality is even more pronounced in explicitly parallel, EPIC-type machines, such as the Intel Itanium [1], Philips Trimedia [2], and Equator MAP/CA [3].

When targeting such processors, a compiler cannot judge the impact of optimizations on generated code performance using simple metrics such as instruction count. Instead, the optimization

©2005 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

process has to carefully balance a set of performance factors, such as dependence height, register pressure, and resource utilization. More importantly, the compiler must also anticipate dynamic effects such as cache misses and branch mispredictions, and avoid or mask them if possible.

To accomplish this task, a compiler needs a large number of complex transformations. Unlike traditional compiler optimizations, such as dead code elimination or constant folding, few of these transformations are universally beneficial. Most of them constitute tradeoffs, improving some performance factors while down‘grading others. For example, loop unrolling increases instruction-level parallelism (ILP) but may adversely affect cache performance, whereas if-conversion avoids branch stalls but increases the number of instructions that must be fetched and issued. Worse, the final outcome of any code transformation ultimately depends on its interactions with subsequent transformations. For example, a software pipelining transformation that originally seems beneficial may lead to more spills during register allocation, thus worsening performance.

Clearly, a successful optimizing compiler must not only incorporate a rich set of optimization routines, but must also correctly determine where and when to apply each one of them. In today's compilers, this is usually achieved through the use of *predictive heuristics*. Such heuristics examine a code segment right before an optimization routine is applied on it, and try to *a priori* judge the optimization's impact on final performance. Usually, a great amount of time and effort is devoted to crafting accurate heuristics. However, a heuristic's task is complicated not only by the complexity of the target platform, but also by the fact that it must anticipate the effect of the current code transformation on all subsequent optimization passes. To make a successful prediction in all cases, each heuristic would ultimately have to be aware of all other heuristics and optimization routines, and all the ways they might interact. Furthermore, all heuristics would have to be changed every time an optimization routine is added or modified. In today's complex optimizers this is clearly an unmanageable task. Therefore it is to be expected that real-world predictive heuristics will make wrong optimization decisions in many cases.

To manage these complications, compiler writers do not fully specify the heuristic and optimization behavior during compiler development. Instead, they leave several optimization *parameters* open. For example, the maximum unroll factor that a loop unrolling heuristic may use can be such a parameter. Similarly, an if-conversion parameter may control exactly how balanced a branch has to be before if-conversion is considered. The values of such parameters are determined during a tuning phase, which attempts to maximize a compiler's performance over a representative sample of applications. In essence, such parameters give the compiler's components a limited ability to automatically adapt to the target architecture, to the target application set, and to each other.

Parameterization and tuning have proven to be very effective in improving a modern compiler's performance. However, they are still an imperfect answer to modern optimization needs. No matter how sophisticated a tuning process is, the end result is still a single, rigid compiler configuration, which then has to be applied to code segments with widely varying optimization needs. In the end, tuning can only maximize the *average* performance across the sample applications. However, this "one size fits all" approach will unavoidably sacrifice optimization opportunities in many individual cases. This effect is especially pronounced when the compiler is applied on code that is not well represented in its tuning sample.

To address these limitations of traditional compiler organization, *iterative compilation* has been proposed [4, 5, 6]. Instead of relying on *a priori* predictions, an iterative compiler applies many different optimization configurations on each code segment. It subsequently compares the different optimized versions of each segment and decides which one is best *a posteriori*. This allows the

compiler to adapt to the optimization needs of each code segment. Previous research indicates that iterative compilation can provide significant performance benefits.

The problem with current iterative compilation approaches is their brute force nature. Such approaches identify the correct optimization path by considering all, or at least a great number of, possible optimization paths. This incurs prohibitive compile time costs. Therefore, iterative compilation has been currently limited to small parts of the optimization process, small applications, and/or application domains where large compile times are acceptable, such as embedded processors and supercomputing.

This article presents Optimization-Space Exploration (OSE), a novel iterative compilation method. OSE realizes the performance potential of iterative compilation while addressing the applicability limitations of existing approaches. This makes OSE the first iterative compilation method suitable for general-purpose, industrial-strength compilers. More specifically, OSE keeps compile-time requirements reasonable by employing the following key ideas.

- Although predictive heuristics are unable to anticipate the full impact of an optimization routine on final code quality, they still encode valuable information on an optimization's behavior. Using this information, an iterative compiler can make intelligent choices as to which part of the optimization space to explore, reducing the number of different optimization configurations that have to be tried.

- For any given application set, a sizable part of the configuration space causes only modest performance gains. Thus the configuration space can be aggressively pruned during a compiler tuning phase.

- On any given code segment, the performance of different configurations is often correlated. This allows the compiler to utilize feedback to further prune the exploration space at compile time.

- Instead of selecting the best optimized version of the code by measuring actual runtimes, an OSE compiler relies on a static performance estimator. Although this approach is less accurate, it is much faster.

A preliminary version of this article has appeared in the CGO proceedings [7]. In comparison with [7], this work uses a much improved instrumentation system (Section 4.1), which, among other things, allowed us to obtain more accurate results, improve the OSE compiler's tuning, and further refine the OSE setup (especially Sections 3.2 and 5.1.2). This work also includes a much more accurate and in-depth experimental evaluation of the system (especially Sections 4.2 and 5.3 to 5.5), and an expanded presentation of related work (Section 2).

The rest of this article is divided into two parts. The first part elaborates on the ideas discussed above. Section 2 further motivates OSE by discussing the shortcomings of both the conventional optimization approach and of current iterative compilation approaches. Section 3 presents OSE in detail. The second part is devoted to experimental evaluation. Section 4 evaluates the behavior of predictive heuristics in real-world compilers. Section 5 uses a proof-of-concept implementation to evaluate the performance of OSE. Finally, Section 6 concludes.

2. Motivation and Related Work

Typically, modern compilers apply a "one size fits all" approach to optimization, whereby every code segment is subjected to a single, uniform optimization sequence. The only opportunity to customize the optimization sequence to the needs of each individual code segment is offered through predictive heuristics. However, the difficulty of characterizing interactions between optimization routines, as well as the complexity of the target architecture, make it very hard to make accurate optimization decisions *a priori*. As a result, no single compiler configuration allows optimizations to live up to their maximum potential. Although such an optimization process can be tuned for maximum average performance, it will still sacrifice important optimization opportunities in individual cases. Section 2.1 strengthens this argument through a bibliographic survey. Experimental evidence will be provided in Section 4.

Iterative compilation takes a different approach. Instead of relying on *a priori* predictions, an iterative compiler applies many different optimization sequences on each code segment. The different optimized versions of each code segment are then compared using an objective function, and the best one is output. Thus iterative compilation is able to find the "custom" optimization approach that best meets each code segment's needs. Although this approach usually results in significant performance gains, it requires prohibitively large compile times. This has prevented iterative compilation from being broadly applied. Most importantly, this has rendered iterative compilation unsuitable for production compilers. Section 2.2 examines the benefits and shortcomings of previously proposed iterative compilation approaches.

2.1 Predictive Heuristics, Interactions, and Traditional Compilation

Whitfield et al. [8] propose an experimental framework for characterizing optimization interactions, both analytically and experimentally. This allows a compiler developer to examine different ways to organize optimization routines and study how they interact. Studies performed using this framework underscore the fact that optimization routines interact heavily, in ways that are difficult to predict. Ultimately, no single optimization organization is ideal for all cases; each compiler configuration exhibits different strengths and weaknesses.

Other work has focused on addressing particularly problematic optimization interactions and developing better heuristics to circumvent performance pitfalls. Heuristics that try to avoid register spilling due to overly aggressive software pipelining have been proposed [9, 10]. Although the proposed heuristics are quite sophisticated, the authors describe cases that the heuristics cannot capture. Among the best studied optimization interferences are those that occur between scheduling and register allocation. Proposed heuristic techniques seek to minimize harmful interferences by considering these two code transformations in a unified way [11, 12, 13, 14]. Continuing efforts in this area indicate that none of the existing heuristics can fully capture these interferences.

Hyperblock formation and corresponding heuristics have been proposed to determine when and how to predicate code [15]. However, even with these techniques, the resulting predicated code often performs worse than it did originally. In an effort to mitigate this problem, techniques that partly "undo" predication by reinserting control flow have been proposed [16]. This need to reexamine and roll back code transformations underscores the difficulty of making *a priori* optimization decisions.

Recognizing the difficulty in hand-crafting heuristics, [17] lets heuristics "evolve" using genetic algorithms. When the genetic algorithm tries to identify the best overall register allocation heuristic for a given benchmark set, it comes up with a heuristic that leads to a 9% performance improvement.

However, when the genetic algorithm is allowed to converge on a different heuristic for each benchmark, the performance improvement is 23% on average. Thus this work underscores the fact that a "one size fits all" optimization approach, even a well-tuned one, is liable to sacrifice performance.

These works are just a sample of the research done to address problems of predictive heuristics. The continuing effort to design better predictive heuristics and to improve compiler tuning techniques indicates that the problem of determining if, when, and how to apply optimizations is far from solved.

2.2 Iterative Compilation

Cooper et al. [4] propose a compilation framework called *adaptive compilation*, which explores different optimization phase orders at compile time. The results of each phase order are evaluated *a posteriori* using one of several objective functions. This system is experimentally evaluated on a small FORTRAN benchmark. Depending on the objective function selected, the adaptive compilation system can produce a 13% reduction in code size or a 20% reduction in runtime relative to a well-tuned traditional compiler. Although some rudimentary pruning techniques are used, the system still needs from 75 to 180 passes before it can identify a solution within 15% of the ideal one.

The OCEANS compiler group [5] has also investigated iterative compilation approaches, mainly within the context of embedded system applications. An initial study [18] on iterative applications of loop unrolling and tiling on three small numerical kernels proves that the iterative compilation approach can cause up to a fourfold increase in generated code performance. A more realistic study [19], involving three loop transformations applied to more sizable numerical benchmarks, achieves a 10% improvement over an aggressively optimizing traditional compiler. Despite the presence of pruning techniques, the system still needs to apply up to 200 different optimization sequences before this performance gain is achieved.

The GAPS compiler project [6] studies the iterative application of loop transformations on numeric benchmarks for parallel processors. Genetic algorithms are used to guide the search for the best optimization sequence at compile time. When applied on a numeric benchmark, the GAPS compiler is able to produce a 75% performance improvement in comparison to the native FORTRAN compiler. The compile time needed for a single small benchmark is about 24 hours.

In an interesting variant of iterative compilation, Wolf et al. [20] present an algorithm for combining five different high level loop transformations, namely fusion, fission, unrolling, interchanging, and tiling. For each set of nested loops the proposed algorithm considers various promising combinations of these transformations. Instead of fully applying each transformation sequence, the algorithm pre-evaluates them by "simulating" their application using a skeleton of the original code. A performance estimator then selects a single sequence for actual application. When evaluated on scientific code, the proposed algorithm yields a 15% performance improvement over non-iterative approaches. Although no compile-time results are included in that paper, the proposed algorithm seems reasonably efficient. However, the algorithm cannot be generalized to other optimizing transformations or to non-numerical applications since it depends on a thorough understanding of the interactions between these transformations within the numerical application domain.

3. Optimization-Space Exploration

Previous research described in Section 2.2 amply demonstrates the promise of iterative compilation. However, current iterative compilation approaches have excessive compile-time requirements and limited applicability. This motivates us to propose Optimization-Space Exploration (OSE), a compilation methodology that realizes the performance potential of iterative compilation while being practical enough for the general-purpose domain.

Like other iterative compilation approaches, OSE applies different optimization configurations to each code segment. The final decision about which optimization configuration performs best is taken *a posteriori*, that is after the resulting optimized versions of the code have been examined. However, OSE differs from other iterative compilation approaches in several crucial ways.

- Predictive heuristics are used in order to limit the optimization options explored.
- OSE eliminates redundant optimization configurations by aggressively pruning the configuration space during a compiler tuning phase.
- OSE employs feedback in order to dynamically prune the search space at compile time.
- A fast static estimator is used to compare the relative performance of different optimization paths.

The remainder of this section examines each of the above ideas in greater detail. The section concludes with a brief discussion of OSE in the context of dynamic optimization and managed runtime environments.

3.1 Limiting the Configuration Pool through Predictive Heuristics

As noted in previous sections, predictive heuristics are unable to anticipate the full impact of a code transformation on final code quality. However, well-crafted heuristics still encode valuable information about a code transformation's behavior. OSE takes advantage of this fact in order to limit the number of different optimization configurations that need to be explored.

Consider loop unrolling as an example. For every loop, an iterative compiler would have to try a great number of different loop unrolling factors. OSE takes a different approach. A well-crafted and well-tuned loop unrolling heuristic, like the one found in a high-performance traditional compiler, is expected to identify the correct loop unrolling factor for a fair number of cases. To capture the remaining cases, configurations containing different variants of the original heuristic can be applied. For example, some such variants could restrict the maximum loop unrolling factor allowed. Configurations that forgo loop unrolling entirely can also be tried.

By trying many variants of each optimization's heuristic, OSE correctly captures the optimization needs of many more code segments than a traditional compiler. Of course, since heuristics are imperfect, this approach cannot capture *all* cases, like an exhaustive iterative compiler can. This, however, is a worthy tradeoff when considering compile-time savings. For example, the iterative compiler proposed in [19] has to consider 16 different unroll factors for each code segment. In comparison, the OSE prototype presented in Section 5 only considers 4 different loop unrolling heuristics. For every optimization, exploiting predictive heuristics causes a similar reduction in the number of choices under consideration. This leads to an overall reduction of the configuration space by a factor exponential in the number of optimization passes.

In subsequent discussions, any variable that controls the behavior of a heuristic or an optimization routine will be referred to as an optimization *parameter*. A full assignment of values to all parameters in an optimizing compiler forms an optimization *configuration*. The set of all configurations that an OSE compiler has to explore constitutes the *exploration space*.

3.2 Static Configuration Selection

By exploiting heuristics, an OSE compiler has to explore a smaller configuration space than that of an exhaustive iterative compiler. However, the size of this space is still prohibitively large. Since every parameter can take at least two values, the total size of the exploration space is exponential with regard to the number of available parameters. Clearly, exploring this space in its entirety at compile time would be impractical. Therefore a radically reduced configuration set has to be selected statically, that is during the OSE compiler's tuning.

Static configuration selection exploits the fact that configurations are not equally valuable. Certain configurations may perform badly in the vast majority of cases. For example, such would be the case of configurations that inline over-aggressively on systems with small instruction caches. Such configurations can be omitted with little performance loss. Other configurations may form clusters that perform similarly in most cases. For example, on a processor with limited resources and small branch misprediction penalties, configurations differing only on if-conversion parameters would fall under this category. In such cases, keeping only one representative configuration from each cluster and pruning the rest would not lead to significant performance losses.

More formally, the goal of the static pruning process is to limit the configuration space to a maximum of K configurations with as little performance loss as possible. The performance of a configuration set is judged by applying it to a set of representative code samples S. The exact value of K is dictated by compile time constraints.

Ideally, the static selection algorithm would determine the best configuration space of size K by considering all possible combinations of K configurations. However, typical exploration spaces are so big that a full consideration of them is impractical, even during compiler tuning. For example, the full exploration space of the OSE prototype described in Section 5 contains 2^{17} configurations, and its full traversal would take roughly 45 years. Therefore, the static selection algorithm has to rely on a partial traversal of the configuration space, even though this may lead to suboptimal results.

The OSE static selection algorithm consists of an *expansion* step and a *selection* step, repeatedly applied until new steps do not provide significant new benefits, or until a time limit is reached.

Beginning with a set of configurations C_S, used as seeds, the expansion step constructs the set C_E of all configurations differing from one of the seeds in only one parameter. Subsequently, every configuration in C_E is applied on every code sample in S, and the runtimes of the optimized codes thus produced are measured.

Subsequently, the selection step determines the K-element subset of C_E that maximizes the performance of OSE. For that purpose, the "exploration performance" of each such subset is determined, as follows: Let $R(s, c)$ be the runtime of a code sample s when optimized using an optimization configuration c. Then the exploration value of a set of configurations C on a set of code samples S is given by the formula:

$$EP(C, S) = \sqrt[|S|]{\prod_{s \in S} \min_{c \in C} R(s, c)}$$

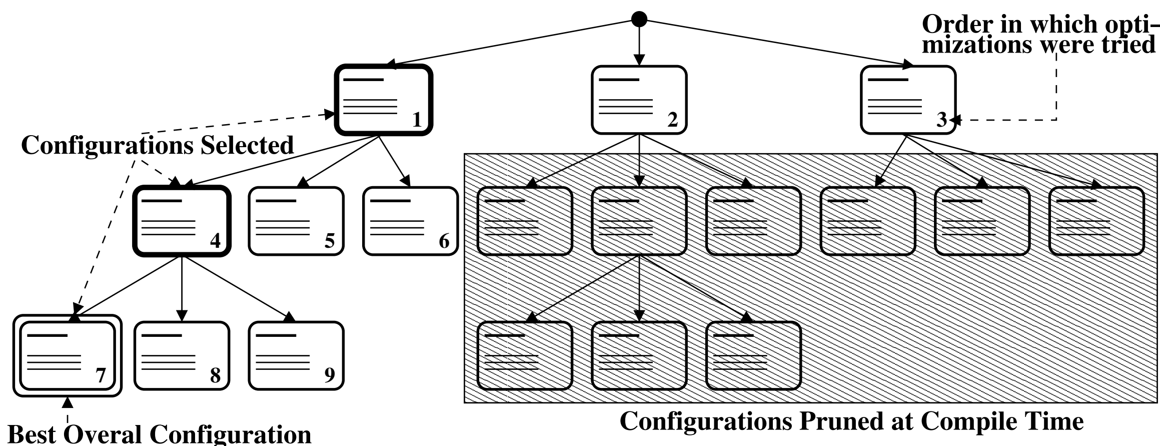

Figure 1: Automatically generated search tree annotated based on a hypothetical run of OSE.

That is, we calculate the geometric mean of the best-performing version of each code sample produced by a configuration in C. The selection step simply determines the exploration value of all K-element subsets of C_E, and selects the one with the best (that is, lowest) exploration performance. The configurations in the set thus selected become the new seeds, on which the expansion step is applied again, and so forth.

The effectiveness of the above process depends greatly on the choice of the initial seeds. A bad choice of seeds may lead to slower convergence, trap the algorithm in local minima, or both. It is therefore important to start with a configuration that is known to perform well on average. Such a configuration would roughly correspond to the optimization process of a well-tuned non-iterative compiler.

3.3 Feedback-directed Compile-time Pruning

By exploiting feedback, an OSE compiler can dynamically prune the exploration space at compile time. On any given code segment the compiler can begin by applying a small set of configurations. Feedback on how these initial configurations performed can help the compiler make an informed choice on which configurations to try next. Feedback from these configurations can be used to select the next set to be tried, and so on. Thus only a portion of the configuration space needs to be explored for each code segment at compile time.

This approach works because different configurations are generally *correlated*. In general, a given configuration performs well on code segments that exhibit certain *code properties*. In many cases, the code properties required by two different configurations may overlap, whereas in other cases they may be unrelated. For example, a configuration that emphasizes software pipelining will perform well on code segments containing small, straight-line loops with complicated dependence chains. On the same code segments, a configuration applying loop unrolling is also likely to perform well. On the other hand, a configuration that forgoes loop optimizations is likely to underperform. Consequently, if an OSE compiler finds out that software pipelining performs well on a code segment, it can decide to try loop unrolling configurations, while forgoing configurations that do not concentrate on loops.

The OSE compiler can exploit configuration correlations by organizing the set of K configurations, as determined in the static selection phase, into a tree, as shown in Figure 1. For each code segment, configurations at the top level of the tree are tried first, and the best one is selected. Subsequently, the children of the selected configuration are tried. After the best of these configurations is selected, its children are in turn tried, and so on. After the bottom of the tree is reached, the best one of the configurations selected at each level is the one that prevails.

This configuration tree has to be constructed during the tuning phase of an OSE compiler. Of course, the notion of code properties is too abstract to be practically useful in this task. However, correlations between configurations can be determined experimentally. Let us assume that an L-way tree is desired. From the K configurations remaining after static pruning, the best-performing combination of L configurations is selected as the top level of the tree. Next the set S of code samples can be partitioned into L subsets, $S_1, S_2, \ldots, S_L$. Subset S_i contains the code segments for which the i'th configuration, c_i, outperforms the other top-level configurations. For each i, the L most valuable configurations for the limited code sample set S_i are then be determined. Essentially, these are the configurations that are most likely to succeed on code segments that respond well to c_i. Therefore, these configurations become the children of c_i in the tree. Subsequent tree levels can be formed by repeating the same process on each set of siblings.

3.4 Performance Estimation

Ideally, an OSE compiler would select the best-performing version of each code segment by measuring actual runtimes. Since code segments cannot be run in isolation, the whole program would have to be compiled before the performance of a single version of a single code segment could be evaluated. Furthermore, the performance of each code segment is dependent not only on its own features, but also on the features of other code segments in the program. This is due, among other things, to cache and branch prediction effects. Therefore, an absolutely accurate judgment on a code segment's performance would have to be obtained through running it in conjunction with every other possible combination of optimized versions of all other code segments in the program. This approach is clearly impractical.

Instead, an OSE compiler makes performance judgments using a static performance estimator. Such an estimator can make predictions based on a simplified machine model and on profile data. In general, obtaining a static prediction of a code segment's runtime performance is a non-trivial task [21]. However, the job of an OSE performance estimator is much simpler, because it only needs to provide a *relative* performance prediction. Rather than trying to determine the exact runtime of a code segment, this estimator has to compare two code segments and predict which one is faster. Moreover, since the code segments compared will actually be differently optimized versions of the same source code, they will be generally similar, and differ in only one or two crucial ways. If, for example, the estimator has to choose between two differently unrolled versions of the same original loop, one of the two versions will have a better schedule, whereas the other will have a smaller code size, whereas all their other features will be very similar. Thus the estimator's task will come down to weighing the scheduling gains versus the code size expansion. Thus the OSE estimator is able to make mostly accurate predictions by simply scoring different code segments according to several performance indicators, such as static cycle count, code size, and memory access patterns. Of course, the exact form of these performance indicators and their relative weight depends on the

target architecture and the target application domain. A concrete OSE performance estimator for the Itanium architecture will be presented in 5.

3.5 Dynamic OSE

As presented up to now, OSE is primarily a static compilation method. However, there are obvious ways in which OSE could be implemented in, and benefit from, dynamic optimization platforms (DOPs) and managed-runtime environments (MRTEs). Indeed, such environments would make OSE more successful and accurate by reducing its reliance on profile data and by eliminating the inaccuracies inherent in static performance estimation.

To respect the generally tighter timing constraints of MRTE compilation, an OSE implementation on such a system would first compile all procedures in the traditional way. Using a lightweight instrumentation, like the one described in 4.1, the system could then accumulate the execution time spent on each procedure. Once a procedure's accumulated runtime exceeds a predetermined limit, the system would judge that this procedure is "hot", and thus worthy of further optimization. It would then proceed to produce differently optimized versions of the procedure using the configurations in the first level of the compile-time tree (Section 3.3). Calls to the original procedure would be redirected to a simple harness, which would satisfy each call by randomly invoking one of these versions. After some time, the system will have gathered enough execution time statistics to know which of these versions performs best. That version would then replace the original function's code. If the program continues spending a significant amount of execution time on this procedure, then the system could repeat the above process with the second level of the tree, and so forth.

In addition to DOPs and MRTEs, which perform optimization during a single invocation of a program, continuous optimization environments (COEs), which can gather statistics and re-optimize throughout an application's deployment lifetime, have been proposed [22]. A COE would actually be the ideal environment for OSE. In addition to applying OSE dynamically as described above, the COE could leverage repeated runs of the application over an extended time period in order to further explore the full configuration space, thus overcoming any suboptimal choices made during static configuration selection. One way to do this would be to obtain random points of the configuration space, as seen in Section 5.5, and use the best-performing ones as new seeds for the expansion-selection sequence of Section 3.2. The results of this process could be communicated to the original OSE compiler, in order to enhance its performance on non-COE applications and to provide a better starting point for future COE runs.

4. Evaluating Predictive Heuristic Failure

Section 2.1 argued that any traditional compiler, no matter how well-tuned, is bound to sacrifice performance opportunities due to incorrect optimization decisions. This section quantifies this performance loss in real-world compilers. We begin with a detailed presentation of the experimental setup, which is based on the Intel Electron compiler, in Section 4.1. This setup will be used again in subsequent sections. Section 4.2 presents experimental results on Electron. Finally, Section 4.3 briefly surveys the behavior of predictive heuristics in other compilers.

Parameter	Meaning
Optimization level	-O2 (default) performs standard optimizations, including loop optimizations [23]. -O3 performs all -O2 optimizations plus riskier optimizations that may degrade performance, including aggressive loop transformations, data prefetching, and scalar replacement.
HLO level	Same as above, but affects only the high-level optimizer.
Microarchitecture type	Optimize for Itanium (default) or Itanium 2. Affects the aggressiveness of many optimizations.
HLO after loop norm.	Off by default. Forces HLO to occur before loops are normalized, effectively disabling some optimizations.
Loop unroll limit	Maximum loop unrolling factor. Values tried: 0, 2, 4, 12 (default).
Update dependences after unrolling	On by default. If disabled, it effectively limits optimization aggressiveness on unrolled loops.
Load/store coalescing	On by default. Forms single instructions out of adjacent loads and stores.
Software pipelining	On by default.
Software pipeline outer loops	Off by default.
Software pipelining if-conversion heuristic	On by default. Uses a heuristic to determine whether to if-convert a hammock in a loop that is being software pipelined. If disabled, every hammock in the loop is if-converted.
Software pipeline loops with early exits	On by default.
If-conversion	On by default.
Non-standard predication	Off by default. Enables predication for if blocks without else clauses.
Pre-scheduling	On by default. Runs scheduling before *and* after register allocation. If disabled, runs scheduling only after register allocation.
Scheduler ready criterion	Percentage of execution-ready paths an instruction must be on to be considered for scheduling. Values tried: 10%, 15% (default), 30%, and 50%.

Table 1: Electron optimization parameters used in this article's experiments.

4.1 Experimental Setup

The following study is based on the Intel C and C++ compiler for the Itanium Processor Family (IPF), also known as Electron. Since Electron is the SPEC reference compiler for IPF, it provides a credible experimentation base. Also, IPF is an especially interesting target architecture, since its explicit parallelism and its complicated performance features make the proper application of aggressive optimizations crucial to achieving good performance. For our experimental baseline we used Electron version 6.0 invoked with the command-line parameters `-O2 -ip -prof_use`, which enable intraprocedural optimization, interprocedural optimization and analysis, and profile information use. This is very close to the compiler configuration used to report the official SPEC numbers for Itanium. Note that the `-O2` option is used instead of the more aggressive `-O3`, both for the official SPEC measurements and for our baseline. This is because the more aggressive optimization settings enabled by `-O3` often cause significant performance degradation instead of improvement. This makes a study of Electron's optimization decision failures all the more interesting.

For this study, the behavior of several Electron optimizations was varied and the impact of these variations on compiled code performance was observed. The full list of optimization parameters studied is given in Table 1. The different variations of Electron were applied to the following benchmarks:

- The SPECint2000 benchmarks `164.gzip`, `175.vpr`, `176.gcc`, `181.mcf`, `186.crafty`, `197.parser`, `253.perlbmk`, `254.gap`, `255.vortex`, `256.bzip2`, and `300.twolf`.

- the SPECcfp2000 benchmarks 177.mesa, 179.art, 183.equake, and 188.ammp.
- the SPECint95 benchmarks 099.go, 124.m88ksim, 129.compress, and 132.ijpeg.
- the MediaBench benchmarks adpcmdec, adpcmenc, epicdec, epicenc, g721dec, g721enc, jpegdec, and jpegenc.
- the parser generator yacc

Note that 252.eon is missing from the SPECint2000 benchmarks above. This is because our experimental support programs could not handle C++. More benchmarks are missing from the SPEC95 and MediaBench suites. Quite a few of these benchmarks did not compile or run on Itanium since they were written for 32-bit architectures. For others, the compilation process failed for certain configurations tried. This is not surprising, since this experiment exercises parts of Electron's optimizer that are not intended for general use.

For the performance measurements, executables were run on unloaded HP i2000 Itanium workstations running Red Hat Linux with kernel version 2.4.18. Execution times were obtained using the performance counters of the IPF architecture with the help of the libpfm library [24]. Because most of our experiments required measuring the cycles consumed by each procedure in a benchmark, we developed an instrumentation system that directly manipulates the assembly language produced by Electron, adding appropriate actions at each procedure entry and exit. These actions involve reading and resetting IPF's performance counters and accumulating each procedure's cycle count in special memory locations. These cycle counts are printed to a file by an exit function, installed through atexit(). Whenever whole program runtimes are reported, these are taken to be the sum of the cycles spent in a program's source procedures, excluding time spent in system calls and precompiled libraries. Reported cycle counts do not contain cycles spent in the instrumentation system itself. Also, since the instrumentation system works directly on assembly language, it does not disturb any part of the optimization process. Some cache interference is unavoidable, but it is limited to a few bytes of data accessed per function entry or exit. Each benchmark was run enough times to reduce random drift in the measurements to below 0.5%. The times that a benchmark had to be run varied according to the characteristics of the benchmark and its input sets, from 3 for the bigger benchmarks to about 20 for the smaller ones.

4.2 Heuristic Failure Rates

We have argued in Section 1 that no single optimization process, even a well-tuned one, can be appropriate for all codes. Although parameterization and tuning can maximize the average performance of a compiler, they are bound to sacrifice performance in many individual cases. To quantify the frequency of predictive heuristic failure in Electron, the following experiment was performed. The optimization parameters appearing in Table 1 were grouped into four broad categories: overall optimization approach (1st to 3rd parameter), load/store handling (7th parameter), predication (12th and 13th parameters), software pipelining (8th to 11th parameters), other loop optimizations (4th to 6th parameters), and scheduling (14th and 15th parameters). For each one of these categories we determined how often a non-standard setting of the category's parameters produces noticeably faster code than the default setting. For this purpose we tried each possible parameter setting for each category on a group of code samples comprising the most important procedures in our benchmark set, namely all procedures that consume at least 10% of their benchmark's runtime. There are 66 such functions in our benchmark set.

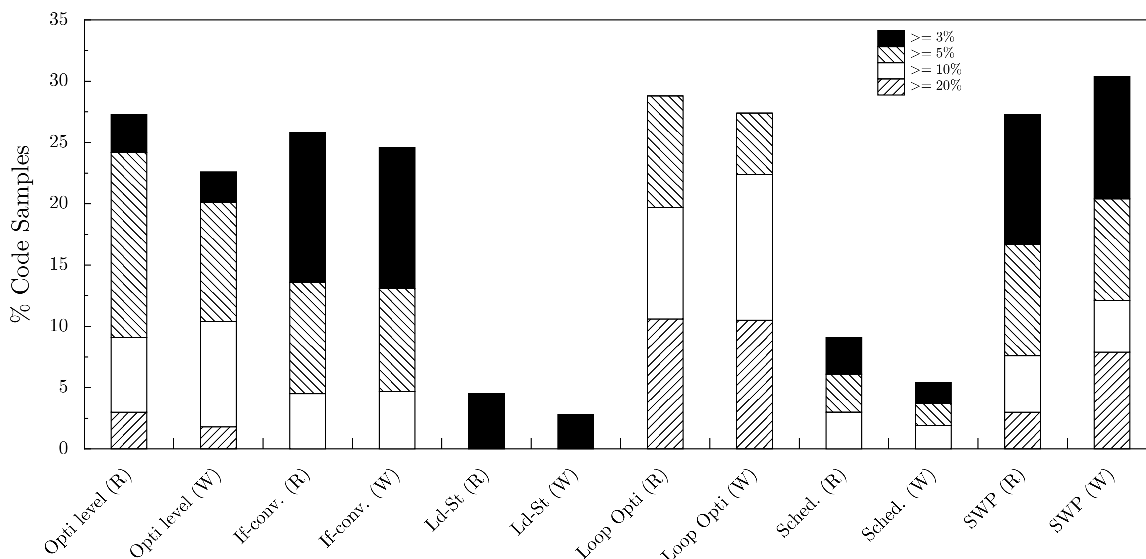

Figure 2: Percentage of procedures for which a non-default setting of a category's parameters causes a speedup greater than 3%, 5%, 10%, and 20% over Electron's default configuration.

The results of this experiment can be seen in Figure 2. For each category, this graph shows how often a non-default parameter setting results in at least 3%, 5%, 10%, and 20% better performance than Electron's default setting for the category's parameters. For columns marked with "R" all procedures count the same, whereas for columns marked with "W" procedures are weighed by their execution weight in the corresponding benchmark.

As we can see in Figure 2, the default setting in each category performs well in a majority of cases. However, a significant number of procedures is not served well by the compiler's default configuration. For example, one of every four procedures could improve its performance by at least 5% if the overall optimization approach were set to different parameters. Similarly, one out of every five procedures would be at least 10% faster if the loop optimizations were customized to its needs. These results provide evidence that even a well-tuned single-path compilation process cannot fit all codes, thus leaving significant performance benefits unrealized. The performance of the OSE-enhanced Electron prototype, seen in Section 5, will provide another, less direct evidence of this fact.

4.3 Heuristics in Other Compilers

The failure of heuristic-driven compilation to make correct optimization decisions is not a phenomenon peculiar to Electron. In one small experiment we varied the loop unrolling factor used by the IMPACT compiler [25] incrementally from 2 to 64. The benchmark `132.ijpeg` performed best for the default loop unrolling factor of 2. However, a performance increase of 8.81% was achieved by allowing each function in `132.ijpeg` to be compiled with a different loop unrolling factor. In a bigger experiment involving 72 different configurations, the individually best config-

urations for `130.li` and `008.espresso` achieved 5.31% and 11.74% improvement over the globally best configuration respectively.

As noted in Section 2.1, the experimental compiler used in [17], which is based on Trimaran, exhibits similar behavior. Allowing the register allocation strategy to be customized on a benchmark by benchmark basis leads to a 13% performance gain over the globally best register allocation strategy.

The experiment described in [26] focuses on GCC targeting Pentium architectures. Reported results show a performance improvement of up to 6% if the compiler configuration is customized on a per-program basis.

5. Evaluation of OSE

In order to evaluate the effectiveness of the OSE approach, we retrofitted the Electron compiler to implement OSE. We call the resulting compiler OSE-Electron. As mentioned earlier, Electron is the SPEC reference compiler for the Itanium platform, thus providing a credible experimental baseline.

5.1 OSE-Electron

This section provides the implementation details of OSE in Intel's Electron compiler for Itanium. This implementation was used to produce the experimental results presented later in this section.

5.1.1 EXPLORATION DRIVER

In the original Electron compiler, optimization proceeds as follows:

1. Profile the code.
2. For each function:
3.   Compile to the high-level IR.
4.   Perform a lightweight high-level optimization (HLO) pass.
5. For each function:
6.   Perform inlining
7.   Perform a second, more comprehensive HLO pass.
8.   Perform code generation (CG), including software pipelining, predication, and scheduling.

In order to build OSE-Electron, we inserted an OSE driver right after inlining (step 6 above). For each procedure the driver decides whether OSE should be applied and which configurations should be tried. Thus the compilation process of OSE-Electron is as follows:

1. Profile the code.
2. For each procedure:
3.   Compile to the high-level IR.
4.   Perform the first HLO pass.
5. For each procedure:
6.   If the procedure is hot:
7.   Perform OSE on the second HLO pass and CG.
8.   Select the procedure's best optimized version for emission.

9. If the function is not hot, use the standard compilation process.

Since OSE-Electron is a retrofit of an existing compiler, it is not an ideal OSE implementation; it incorporates several sub-optimal implementation choices. For example, due to certain technical difficulties the exploration omits the first HLO pass and the inlining process. Also, the exploration is limited to the configuration space described in Table 1. A compiler build for OSE from scratch would make many more optimization parameters available for exploration. Finally, the performance estimator's success suffers from the limited profiling data that Electron makes available, as we will see later in this section.

Although OSE-Electron makes use of the profile weights gathered by the Electron compiler, it is important to note that the OSE technique is not crucially dependent on profile data. Just like any other profile-driven compiler technique, such as inlining or software pipelining, OSE could work with statically determined profile weight estimates.

5.1.2 PERFORMANCE ESTIMATION

Two factors drove the design of the static performance estimation routine in OSE-Electron. The first was compile time. Since the estimator must be run on every version of every function compiled, a simple and fast estimation routine is critical for achieving reasonable compile times. For this reason, the estimator chosen performs a single pass through the code, forgoing more sophisticated analysis techniques. The second limitation resulted from limited information. The final code produced by the Electron compiler is annotated with basic block and edge execution counts calculated in an initial profiling run and then propagated through all optimization phases. Unfortunately, without path profiling information many code transformations make the block and edge profiles inaccurate. Further, more sophisticated profile information, such as branch misprediction or cache miss ratios, could be useful to the estimator, but is unavailable.

Each code segment is evaluated at compile time by taking into account a number of performance indicators. The performance estimate for each code segment is a weighted sum of all such indicators. The indicators used are described below.

Ideal cycle count The ideal cycle count T is a code segment's execution time assuming perfect branch prediction and cache behavior. It is computed by multiplying each basic block's schedule height with its profile weight and summing over all basic blocks.

Data cache performance To account for load latencies, a function of data cache performance, each load instruction is assumed to have an average latency of λ. Whenever the value fetched by a load instruction is accessed within the same basic block, the block's schedule height, used in the computation of T above, is computed using a distance of at least λ cycles between the load-use pair.

Another term is introduced to favor code segments executing fewer dynamic load instructions. The number of load instructions executed according to the profile, L, provides another bias toward better data cache performance.

Instruction cache performance The most obvious predictor of instruction cache performance is of course a segment's code size C. Another performance indicator seeks to bias the estimator against loop bodies that do not fit into Itanium's first-level instruction cache. This is achieved by

the formula:

$$I = \sum_{L \in \text{ loops of } S} \left\lfloor \frac{\text{size}(L)}{\text{size(L1 Icache)}} \right\rfloor \times wt(L)$$

where S is the code segment under consideration and $wt(X)$ is the profile weight of X. The floor operator is used to model the bimodal behavior of loops that just fit in the cache against those that are just a bit too large.

Branch misprediction The Electron compiler does not provide us with detailed branch behavior profile information. Therefore, OSE-Electron has to approximate branch misprediction ratios using edge profiles. For each code segment S, the estimator assesses a branch misprediction penalty term according to the formula:

$$B = \sum_{b \, \in \text{ branches of } S} \min(p_{\text{taken}}, 1 - p_{\text{taken}}) \times wt(b)$$

where p_{taken} is the probability that the branch b is taken, as determined by the edge profiles, and $wt(b)$ is the profile weight of b.

Putting it all together Given a source-code function F, let S_c be the version of F's code generated by a compiler configuration C, and let S_0 be the version of F's code generated by Electron's default configuration. Then the static estimation value for the code segment S_c is computed according to the formula:

$$E_c = \alpha \times \frac{T_c}{T_0} + \beta \times \frac{C_c}{C_0} + \gamma \times \frac{I_c}{I_0} + \delta \times \frac{L_c}{L_0} + \epsilon \times \frac{B_c}{B_0}$$

where terms subscripted with C refer to the code segment S_c, and terms subscripted with 0 refer to the code segment S_0. Whenever two or more versions of a code segment are compared, the one with the lowest estimation value prevails.

A brute-force grid searching method was used to assign values in the interval $[0, 1)$ to the weights $\alpha, \beta, \gamma, \delta$, and ϵ. The same search determined the load latency parameter λ. The grid search used the same sample of procedures that will be used in Section 5.2. The grid search determined the values of α, β, γ, δ, ϵ, and λ that guide the performance estimator to the best possible choices on the sample. The resulting values are: $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.001$, $\delta = 0.03$, $\epsilon = 0.0004$, and $\lambda = 2.6$.

One might assume that the design of an OSE static performance estimator for IA-64 is facilitated by the processor's in-order nature, and that it would be difficult to design similar estimators for out-of-order processors. This, however, is not the case, because of the fact that the OSE performance estimator only needs to make *relative* predictions. Take for example the load-use distance parameter λ above. Although the exact number of stalled cycles because of a cache miss is more difficult to predict on an out-of-order processor, it is still the case that a version of a code segment with greater load-use distances is *less* likely to incur stalls, and thus is preferable. Of course, the exact value of λ would have to be different. However, since parameter values are determined automatically, this would not present a problem to the compiler designer.

5.1.3 HOT CODE SELECTION

To limit compile time, OSE-Electron limits the exploration to the proverbial 10% of the code that consumes 90% of the runtime. For this purpose, the smallest possible set of procedures accounting for at least 90% of a benchmark's runtime is determined. OSE-Electron then applies an OSE compilation process on procedures in this set, and a traditional compilation process on the remaining procedures. We experimentally verified that this fraction yields a good trade-off between compile time and performance by trying a number of other thresholds.

5.2 OSE Tuning

As described in Section 3, an OSE compiler needs to undergo a tuning phase, in which the configuration space is statically pruned, the configuration tree is formed, and the performance estimator is tuned. From the benchmarks described in Section 4.1, we chose to use the SPEC2000 suite as OSE-Electron's tuning set. More precisely, we formed a set of code samples comprising all functions in SPEC2000 benchmarks that consume 5% or more of their benchmark's runtime. There are 63 such procedures in the SPEC2000 suite. The 5% threshold was chosen because timing measurements of procedures with too short runtimes tend to exhibit high levels of noise, which might in turn lead OSE-Electron's tuning phase to wrong choices. Procedure runtimes were obtained by running the SPEC2000 executables, using the instrumentation described in Section 4.1, with the training inputs specified by the SPEC2000 suite. The choice of benchmarks for the tuning set was motivated by the fact that commercial compilers are usually tuned using the SPEC2000 benchmark suite. The rest of the benchmarks mentioned in Section 4.1, which were omitted from the tuning set, will be used later for a fairer evaluation of OSE-Electron's performance.

The parameters described in Table 1 form a space of 2^{17} configurations. From these we selected 25 configurations using the methodology described in Section 3.2. We used Electron's `O2` and `O3` configurations as seeds, and we performed two iterations of the expansion and selection steps. A third iteration was aborted, because its expansion step did not produce any significant performance improvements. These 25 configurations were organized according to the methodology described in Section 3.3 into the 2-level, 3-way tree shown in Figure 3, which contains 12 configurations in all. Finally, the performance estimator described in Section 5.1.2 was tuned using the 63 SPEC2000 procedures in our code sample.

The progress of the tuning phase can be seen in Figure 4. The runtime performance of each benchmark when optimized using Electron's default configuration forms the graph's baseline. The first bar in the graph represents the performance of OSE-Electron at the end of the static selection phase, without static performance estimation or compile-time pruning. Here each procedure in a benchmark is optimized using the 25 configurations produced by the static selection phase, and the best version is selected for emission after measuring actual runtimes. The second bar represents OSE-Electron's performance employing static performance estimation, but no compile-time pruning. For the third bar, both the static estimator and the configuration tree were used. Runtimes of both procedures (in the first bar) and benchmarks were determined by the instrumentation system described in Section 4, using the benchmarks' training inputs. Using the same set of inputs for both tuning and performance measurement allows us to focus on the performance impact of OSE-Electron's features, which might be obscured by input set differences. A fairer evaluation of OSE-Electron, using separate training and evaluation inputs, will be provided in Section 5.3.

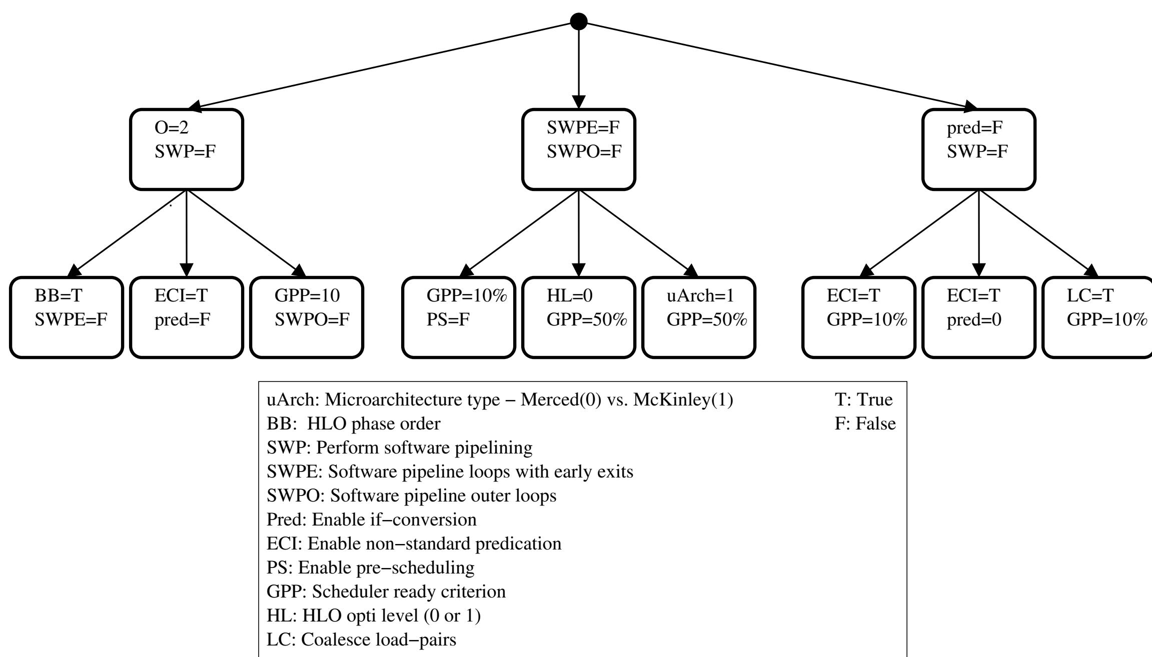

Figure 3: Tree of configurations for OSE-Electron's compile-time search.

As we can see from the graph, OSE-Electron produces a 5.3% overall improvement on the performance of SPEC2000 benchmarks over Electron, IPF's SPEC reference compiler. Gains are especially pronounced for `164.gzip`, `179.art`, and `256.bzip2`. The graph also shows that static performance estimation sacrifices a modest amount of performance. This is inevitable, since static performance predictions cannot always be accurate. Interestingly, in some cases the estimator makes better choices than the actual runtime measurements. This is a result of inter-procedure interactions not taken into account in either experiment, but contributing to the final runtimes. While this adds a factor of uncertainty, note that the average performance improvement due to OSE is well above this factor. These runtime dependences between procedures also explain why OSE-Electron with compile-time tuning can outperform an exhaustive search of the selected configurations.

Figure 4 also shows that the addition of compile-time pruning sacrifices almost no performance. On the other hand, dynamic pruning causes a very significant reduction in OSE-Electron's compile time, as can be seen in Figure 5. This figure compares the compile times of OSE-Electron with and without compile-time pruning. The baseline for this graph is the compile time spent by Electron's default configuration. As we can see, OSE can be applied at a compile-time cost of 88.4% compared to a traditional compiler. For comparison purposes, Electron's default optimizing configuration (`-O2`) is about 200% slower than non-optimizing compilation (`-O0`). Therefore, OSE makes iterative compilation practical enough for the general-purpose domain.

5.3 OSE Performance Evaluation

To obtain a more thorough evaluation of OSE-Electron's performance benefits, we applied it on a set of benchmarks different from its tuning set. For this purpose we used the SPEC95 and MediaBench benchmarks of Section 4.1, as well as `yacc`. The performance improvement caused by OSE-

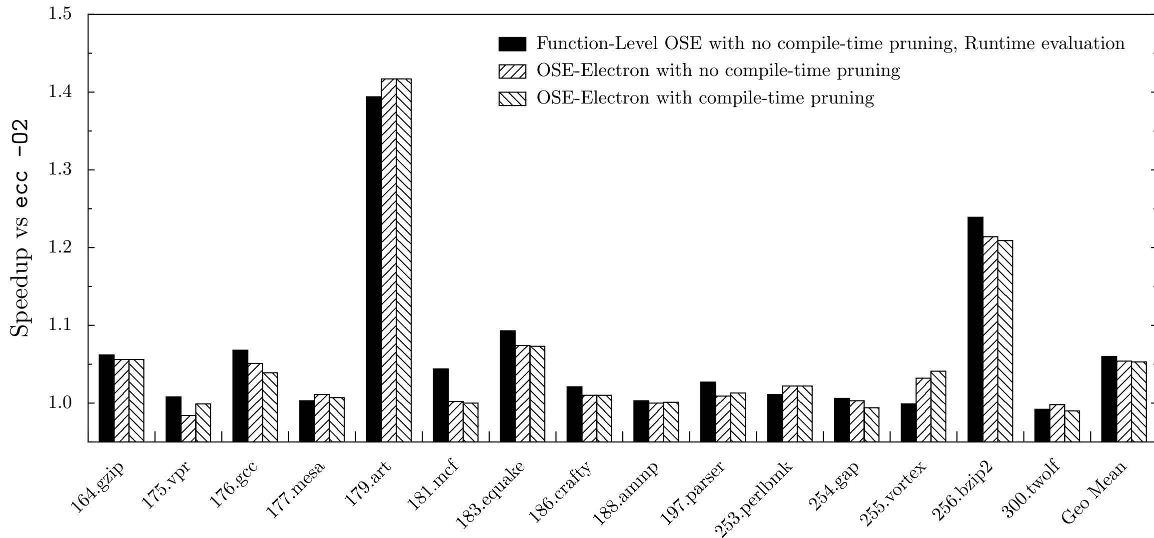

Figure 4: Performance of OSE-Electron generated code for SPEC benchmarks, with and without static performance estimation and compile-time pruning.

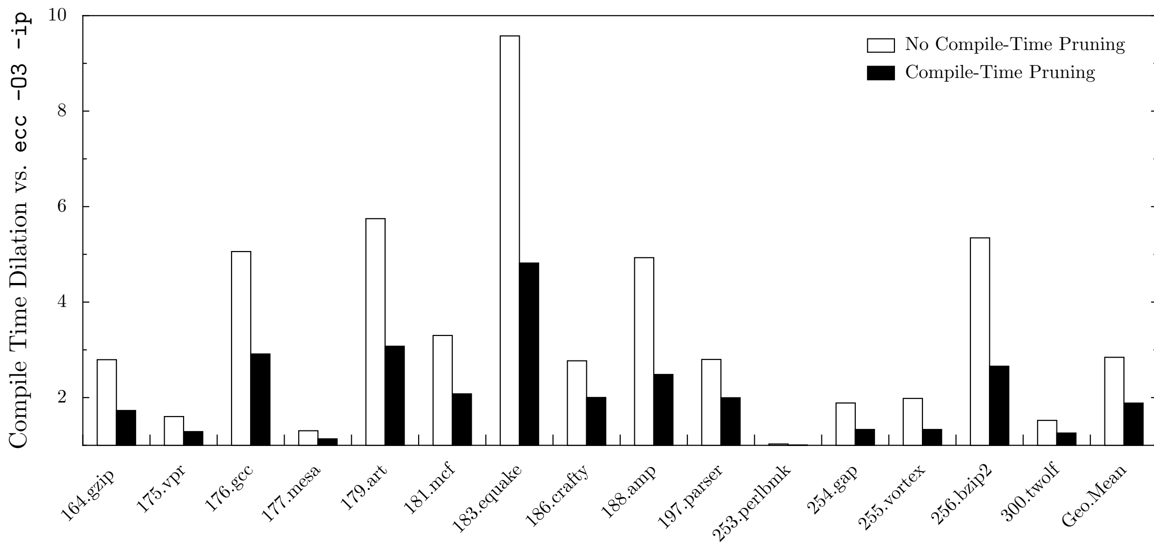

Figure 5: Compile time dilation for OSE-Electron over standard Electron.

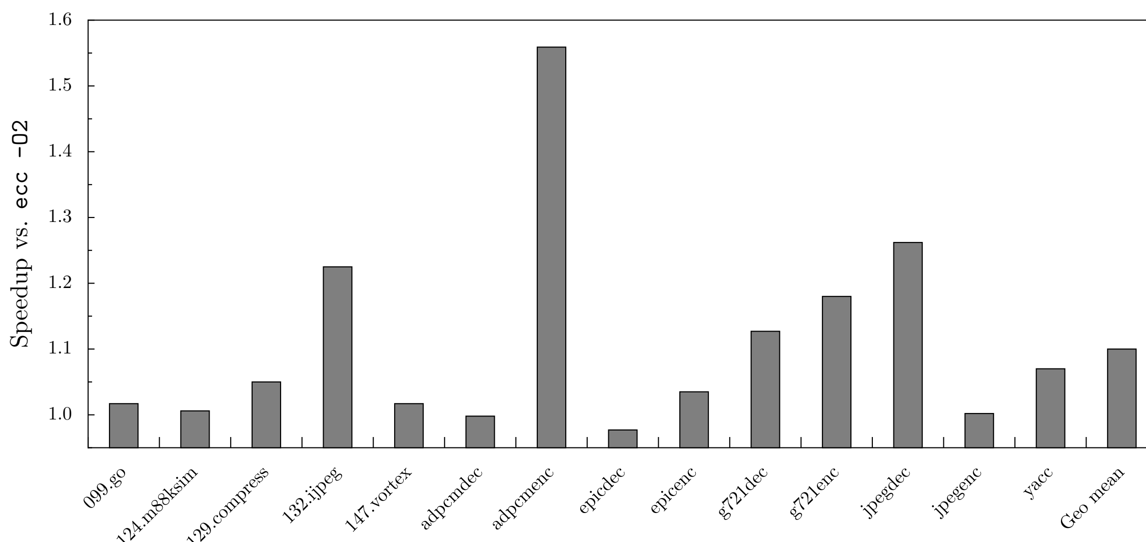

Figure 6: Performance of OSE-Electron generated code for non-SPEC benchmarks.

Electron compared to Electron's default optimizing configuration can be seen in Figure 6. Unlike Figure 4, we used different training and evaluation inputs for each benchmark in this experiment. As we can see, OSE-Electron performs 10% better overall, and up to 56% better in individual cases, than Electron's default configuration.

Counter-intuitively, OSE-Electron performs better on these benchmarks than on the benchmarks in its tuning set. This can be explained by the fact that Electron's heuristics were probably tuned very carefully with the SPEC2000 suite in mind, whereas they were not as well tailored to the benchmarks tried here. OSE-Electron, on the other hand, can fit the optimization needs of both benchmark sets.

5.4 Postmortem Code Analysis

The significant performance benefits produced by OSE in many of the benchmarks tried above motivates us to look for the sources of these benefits. Below we examine three of the most prominent examples of OSE's performance improvements, and identify how the configuration exploration and the performance estimator arrived at these results.

5.4.1 SPECINT95 BENCHMARK `132.ijpeg`

Consider the functions `jpeg_fdct_islow` and `jpeg_idct_islow` in the `132.ijpeg` SPEC95 benchmark. These functions compute forward and inverse discrete-cosine transforms on image blocks. When compiled using Electron's default configuration, these two functions account for about 36% of the benchmark's execution time. Each of these two functions contains two fixed-count loops iterating 64 times.

Electron's high-level optimizer, which is run before the more machine-specific low-level optimizer in its back end, contains a loop unrolling transformation for fixed count loops, controlled by a heuristic. Since the code of the four loops described above contains many data dependencies, which

would prevent efficient scheduling, the loop unrolling heuristic decides to unroll each of these loops 8 times. Subsequently, a second loop unrolling transformation in the back-end optimizer unrolls each loop another 8 times.

While full unrolling seems sensible in this case, if the high-level unrolling is turned off, `jpeg_fdct_islow` sees a 120% performance improvement, with similar results for `jpeg_idct_islow`. This is because complete unrolling makes each function's code bigger than the 16K level-1 instruction cache. The result is that `132.ijpeg` spends 19% of its execution time in instruction-cache stalls when the code in these functions is fully unrolled, and only 5% when unrolling is not applied on them. This instruction cache performance loss overwhelms any gains due to better scheduling. One is tempted to think that better high-level loop unrolling heuristics could avoid this problem. However, this is unlikely, since such heuristics would have to anticipate the usually significant code size effect of all future optimization passes. On the other hand, the OSE performance estimator has the advantage of examining both loop-unrolled and non-loop-unrolled versions of the code at the end of the optimization process, where the problem with loop unrolling is easy to spot.

5.4.2 SPECINT 2000 BENCHMARK `256.bzip2`

Another case where OSE is able to achieve a large performance benefit is the function `fullGtU` in the `256.bzip2` SPEC2000 benchmark. When compiled with Electron's default configuration, this function accounts for 48% of total running time. Our experiments show that a performance improvement of 76% is achieved in this function when software pipelining is disabled.

Software pipelining is applied in order to overlap iterations in a loop while yielding fewer instructions and higher resource utilization than unrolling. During software pipelining, the loop's 8 side exits are converted to predicated code. The conditions for these side exits, and consequently the conditions on the new predicate define operations in the pipelined loop, depend on values loaded from memory within the same iteration of the loop. Since the remainder of the code in the loop is now data-dependent upon these new predicates, the predicate defines are now on the critical path. To reduce schedule height, these predicate defining instructions are scheduled closer to the loads upon which they depend. During execution, cache misses stall the loop immediately at these predicate defines, causing performance degradation.

The performance of this code depends heavily on the ability of the compiler to separate these ill-behaved loads from their uses. However, the constraints governing this separation are difficult to anticipate until after optimization. In this case, the predication causing the problem only occurs after the software pipelining decision has been made. Anticipating and avoiding this problem with a predictive heuristic would be extremely difficult. On the other hand, the OSE compile-time performance estimator can easily identify the problem, since it can examine the load-use distance after optimization.

5.4.3 MEDIABENCH BENCHMARK `adpcmenc`

Examining the adpcmenc benchmark reveals that over 95% of the execution time is spent in one function, `adpcm_coder`. This function consists of a single loop with a variety of control flow statements. With `-O3` turned on Electron aggressively predicates the loop yielding a 12% decrease in schedule height versus `-O2`, which leaves much of the control flow intact. This accounts for all the speedup observed. The OSE-Electron estimator can easily pick the shorter version of the code

since other characteristics considered are similar between the versions. While this fact could lead one to conclude that the `O3` level is simply better than `O2`, changing Electron's default configuration to `O3` would actually lead to performance degradation for more than half the benchmarks in our suite. On the other hand, OSE-Electron is able to deliver the benefits of the `O3` configuration while avoiding its performance pitfalls.

5.5 Evaluating the Pruning Strategy

By following the OSE static selection and compile-time pruning methodologies, OSE-Electron is able to deliver significant performance benefits by trying just 6 configurations per code segment, 3 for each tree level. To evaluate the effectiveness of both these methodologies, we compare the OSE-Electron described above against randomized version of OSE. This version constructs 6 configurations by assigning for each parameter in Table 1 a randomly picked value from the parameter's value set. Each benchmark is then compiled using these random configurations, and the best version of each procedure is selected by using OSE-Electron's static estimator. Figure 7 compares the performance of OSE-Electron with that of its "Monte-Carlo" version.

From the figure we can see that a randomly selected configuration set generally offers less performance benefits than the configuration set picked by OSE-Electron's selection phases. On average, the random configuration set performs about 1% worse than OSE-Electron on SPEC2000 benchmarks, and about 6% worse on the other benchmarks.

Notice that the random configuration set provides big speedups (over 15%) in only 3 benchmarks, whereas the normal OSE-Electron achieves large speedups in 7 benchmarks. The few benchmarks, particularly `256.bzip2`, where a random configuration selection performs better than one would expect occur because the performance improvements in these benchmarks are caused by varying a single optimization parameter: other optimization parameters have very little effect. In these cases each random configuration has a 25% - 50% chance of finding the correct configuration in each random trial. In the experiment above, we try 6 random configurations, meaning that it will find the correct answer with a probability between $1 - (.5)^6$ and $1 - (.25)^6$.

A similar analysis also explains the relatively modest improvement of OSE-Electron versus random configurations on the SPEC benchmark suite. Since Electron was tuned for these benchmarks, many heuristic-controlled configurations do quite well, greatly improving the random configurations' chances of generating good results. Notice that for non-SPEC benchmarks, OSE-Electron significantly out-performs the random configurations. In short, OSE is even more effective when the compiler encounters codes for which it was not tuned.

6. Conclusion

In this article we experimentally demonstrate that predictive heuristics in traditional, single-path, "one size fits all" compilation approaches sacrifice significant optimization opportunities, thus motivating iterative compilation. We then propose a novel iterative compilation approach, called Optimization-Space Exploration (OSE), that is the first such approach to be both general and practical enough for modern aggressively optimizing compilers targeting general-purpose architectures.

Unlike previous iterative compilation techniques, the applicability of OSE is not limited to specific optimizations, architectures, or application domains. Furthermore, OSE does not incur the prohibitive compile-time costs of other iterative compilation approaches. OSE achieves this by leveraging existing predictive heuristics, by carefully selecting the search space during compiler

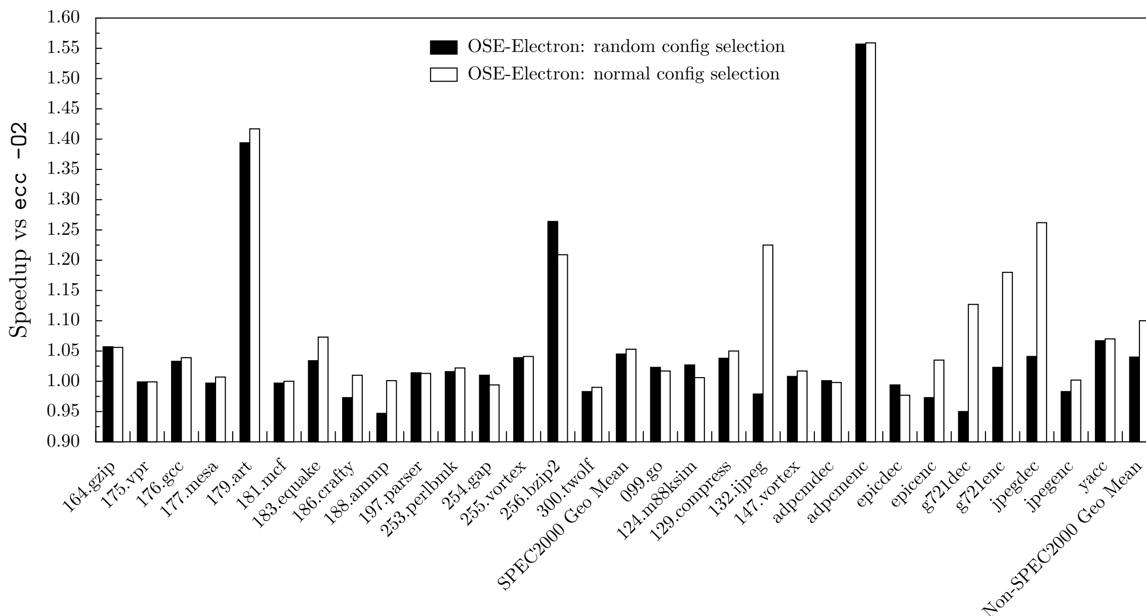

Figure 7: Comparison of OSE-Electron's static and compile-time configuration selection vs. random configuration selection.

tuning, and by utilizing feedback in order to further prune the search space at compile time. Further, OSE employs a fast static performance estimator, thus obviating the need to run and measure multiple optimized versions of the code. Finally, OSE is only applied to the frequently executed portion of the program.

The potential of OSE has been experimentally demonstrated by implementing an OSE-enabled version of Intel's aggressively optimizing production compiler for Itanium. Experimental results from this prototype confirm that OSE is capable of delivering significant performance benefits while keeping compile times reasonable.

Acknowledgements

The authors wish to thank Carole Dulong, Daniel Lavery, and the rest of the Electron Compiler Team at Intel Corporation for their help during the development of OSE and this paper. We also thank Mary Lou Soffa, John W. Sias, and the anonymous reviewers of the CGO conference and the Journal of ILP for their insightful comments. This work was supported by National Science Foundation grants CCR-0082630 and CCR-0133712, a grant from the DARPA/MARCO Gigascale Silicon Research Center, and donations from Intel.

References

[1] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.

[2] Philips Corporation, “Philips Trimedia Processor Homepage,” 2002. http://www.semiconductors.philips.com/trimedia/.

[3] Equator Corporation, “Equator MAP architecture,” 2002. http://www.equator.com/products/MAPCAProductBrief.html.

[4] K. D. Cooper, D. Subramanian, and L. Torczon, “Adaptive optimizing compilers for the 21st century,” *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, 2002.

[5] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoggerbrugge, P. Hu, W. Jalby, P. M. W. Knijnenburg, M. F. P. O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. Stohr, M. Verhoeven, and H. A. G. Wijshoff, “OCEANS: Optimizing compilers for embedded applications,” in *European Conference on Parallel Processing*, pp. 1351–1356, 1997.

[6] A. P. Nisbet, “GAPS: Iterative feedback directed parallelisation using genetic algorithms,” in *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, (Paris, France), October 1998.

[7] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, “Compiler optimization-space exploration,” in *Proceedings of the '03 International Symposium on Code Generation and Optimization*, pp. 204–215, March 2003.

[8] D. L. Whitfield and M. L. Soffa, “An approach for exploring code improving transformations,” *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 1053–1084, November 1997.

[9] J. Llosa, M. Valero, E. Ayguade, and A. Gonzalez, “Modulo scheduling with reduced register pressure,” *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 625–638, 1998.

[10] R. Govindarajan, E. R. Altman, and G. R. Gao, “Minimizing register requirements under resource-constrained rate-optimal software pipelining,” in *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[11] R. Leupers, “Instruction scheduling for clustered VLIW DSPs,” in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, October 2000.

[12] J. R. Goodman and W. C. Hsu, “Code scheduling and register allocation in large basic blocks,” in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[13] D. G. Bradlee, S. J. Eggers, and R. R. Henry, “Integrating register allocation and instruction scheduling for RISCs,” in *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, 1991.

[14] W. G. Morris, “CCG: A prototype coagulating code generator,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Toronto, ON, CA), pp. 45–58, June 1991.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[16] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *International Symposium on Microarchitecture*, pp. 92–103, 1997.

[17] M. Stephenson, U.-M. O'Reilly, M. C. Martin, and S. Amarasinghe, "Genetic programming applied to compiler heuristic optimization," in *Proceedings of the European Conference on Genetic Programming*, (Essex, UK), April 2003.

[18] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff, "A feasibility study in iterative compilation," in *International Symposium on High Performance Computing*, pp. 121–132, 1999.

[19] G. Fursin, M. O'Boyle, and P. Knijnenburg, "Evaluating iterative compilation," in *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing*, (College Park, MD), July 2002.

[20] M. Wolf, D. Maydan, and D. Chen, "Combining loop transformations considering caches and scheduling," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 274–286, December 1996.

[21] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," *Design Automation of Electronic Systems*, vol. 4, no. 3, pp. 257–279, 1999.

[22] T. Kistler and M. Franz, "Continuous program optimization: Design and evaluation," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 549–566, 2001.

[23] Intel Corporation, *Electron C Compiler User's Guide for Linux*, 2001.

[24] "Perfmon: An IA-64 performance analysis tool." http://www.hpl.hp.com/research/linux/perfmon.

[25] 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.

[26] "An evolutionary analysis of GNU C optimizations." http://www.coyotegulch.com/acovea/index.html.