ACME: Adaptive Compilation Made Efficient *

Keith D. Cooper, Alexander Grosul, Timothy J. Harvey,
Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman

Computer Science Department
Rice University
Houston, TX 77005
{keith|grosul|harv|sreeves|devika|linda|waterman}@rice.edu

Abstract

Research over the past five years has shown significant performance improvements using a technique called *adaptive compilation*. An adaptive compiler uses a compile-execute-analyze feedback loop to find the combination of optimizations and parameters that minimizes some performance goal, such as code size or execution time.

Despite its ability to improve performance, adaptive compilation has not seen widespread use because of two obstacles: the large amounts of time that such systems have used to perform the many compilations and executions prohibits most users from adopting these systems, and the complexity inherent in a feedback-driven adaptive system has made it difficult to build and hard to use.

A significant portion of the adaptive compilation process is devoted to multiple executions of the code being compiled. We have developed a technique called *virtual execution* to address this problem. Virtual execution runs the program a single time and preserves information that allows us to accurately predict the performance of different optimization sequences without running the code again. Our prototype implementation of this technique significantly reduces the time required by our adaptive compiler.

In conjunction with this performance boost, we have developed a graphical-user interface (GUI) that provides a controlled view of the compilation process. By providing appropriate defaults, the interface limits the amount of information that the user must provide to get started. At the same time, it lets the experienced user exert fine-grained control over the parameters that control the system.

Categories and Subject Descriptors D.3.4 [*Compilers; Optimization*]: Adaptive compilation in an optimizing compiler

General Terms Experimentation, Performance

Keywords Adaptive compilation

* This work has been supported by Los Alamos Computer Science Institute and by the National Science Foundation through grant CCR0205303. This work does not represent the official opinion of either agency.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

1. The evolution of adaptivity

After decades of research into efficient methods of data-flow analysis and the development of a plethora of transformations, we began to ask the question: how effective are our compilers? The literature is replete with evidence of the efficacy of individual transformations, but the issue of combining the correct set of optimizations for a wide variety of input codes is a problem both recognized and generally ignored. Reasoning about the interactions between transformations is dauntingly complex, and the cost of measuring these interactions quantitatively was, until recently, prohibitive. For example, our first attempts to enumerate these interactions empirically required fourteen CPU months for a relatively small set of optimizations. Some batch compilers, such as the VISTA system by Kulkarni *et al.*, address the problem by repeatedly running their entire suite of optimizations in a round-robin fashion until the code stops changing [20].

The increases in processor speed have enabled experiments that use the computer itself to explore different combinations and permutations of optimization sequences. In short, these experiments have shown that hand-picked optimization sequences do not consistently use the compiler to its greatest advantage [8, 9, 25]. Indeed, in the VISTA system, Kulkarni *et al.* report improvements over their fixed-order, round-robin compilation when they added adaptivity into their system [19].

These results would be interesting even if they simply identified a maximally performing sequence of optimizations, but these experiments have also shown that different input codes benefit from remarkably different sequences. This second result argues strongly for a compiler that can change its behavior for each input program.

The compiler can adapt in two ways. First, it can analyze the input code to detect features amenable to specific transformations and invoke the corresponding optimizations [27, 28]. In the case of opportunities across multiple transformations, however, this is currently beyond our capabilities: we do not yet know how to identify, in general, the salient characteristics of the input code that contribute to performance differences of sequences of transformations, nor do we have a vocabulary to describe the interaction between transformations which have markedly different effects on the code.

The second method to find a good optimization sequence is the one we currently employ. Feedback-driven adaptive compilation starts by compiling the code with some sequence of optimizations. The adaptive system then runs the code to produce a measurement. It evaluates the measurement and instructs the compiler to recompile the code using a modified sequence of optimizations. We have experimented with different methods for guiding the compiler, including genetic algorithms, greedy algorithms, hill-climbing algorithms, and random probing. We have reduced the number of evaluations needed to find a good sequence from $10,000$ in our initial

experiments to somewhat less than 500, using different methods of searching the space of sequences [8, 7, 15]. These methods produce consistent and significant improvements in code quality, and our experiments in known subspaces suggest that the methods find optimization sequences that are close to the best that can be found for a given set of transformations.

Even with our success in identifying efficient search methods, the expense is still prohibitive for most users. 500 compilations and executions can take hours for a moderate sized program.

ACME addresses the performance problem by reducing the number of executions to a single profiling run at the start of the adaptive compilation. We use the profiling data in the analysis phase of the adaptive compiler to perform *virtual execution* (explained in Section 4), a method of performance estimation based on instruction counts. We have used ACME to compare the runtime of the compilation with and without virtual execution; Section 5 shows that virtual execution drastically reduces overall compilation time in our adaptive system.

A second hindrance to widespread adoption of adaptive compilation is the complexity of the interface. Our system, ACME, can be invoked from the command line. It enables four different search algorithms with different sets of parameters, sixteen different optimizations, *etc*. As such, a single invocation of ACME can require as many as fourteen different parameters.

We believe an adaptive system is unlikely to gain widespread use if the interface is not designed around the users' needs, regardless of the practicality of adaptation. We address the complexity of running the compiler with an easy-to-use interface that experience suggests provides correct levels of information to novices and more experienced users alike.

2. Related work

Cooper *et al*. used genetic algorithms to find a good ordering of compiler optimizations to minimize executable size in 1999 [8]. Since then, there has been a great deal of research into using adaptive techniques in compilers to produce better executables. Several researchers have continued to examine the problem of ordering optimizations [1, 9, 20, 19, 25]. Understanding of the problem has increased, and adaptive compilation has led to the production of significantly faster executables. Adaptive compilation has also been successfully used to improve the performance of individual optimizations via parameter selection [17, 24, 28].

Despite the success of this research, adaptive compilation has not been widely adopted. Adaptive compilation's use has been limited by the time required to find a good solution and the usability of the system – the two issues that ACME addresses. Other researchers have also investigated how to make adaptive compilation more practical. Dr. Options is an automatic system that recommends options for the PA-RISC compiler [14]. Dr. Options combines profile information, heuristics, and user input to simplify the process of selecting options. However, the system does not use repeated compilation and evaluation to improve results.

The VISTA system is an interactive system which concentrates on reducing the compilation time, similar to ACME [19]. VISTA reduces the number of executions needed by storing a representation of each compilation and only executing code which has never been seen before. In their results, they run the code only about 15% of the time. Virtual execution takes a radically different approach, executing the code once[1] and thereafter producing a close estimate of the execution cost for variant versions of the code from data gathered during the single execution.

VISTA also uses a number of techniques to weed out compilation sequences that probably will not change the code – this avoids unnecessary invocations of the compiler. This sophisticated analysis solves the other half of the performance problem: in our system, about 26% of the time is spent transforming the code, with the remaining time devoted to linking and running the code. ACME addresses the execution bottleneck, while VISTA's techniques could be used symbiotically to address the compilation bottleneck.

[1] In reality, ACME runs the code twice, once with no optimization to get profiling data for virtual execution, and the second time with our compiler's default sequence of optimizations to get a performance baseline. Clearly, this second run is not necessary for virtual execution.

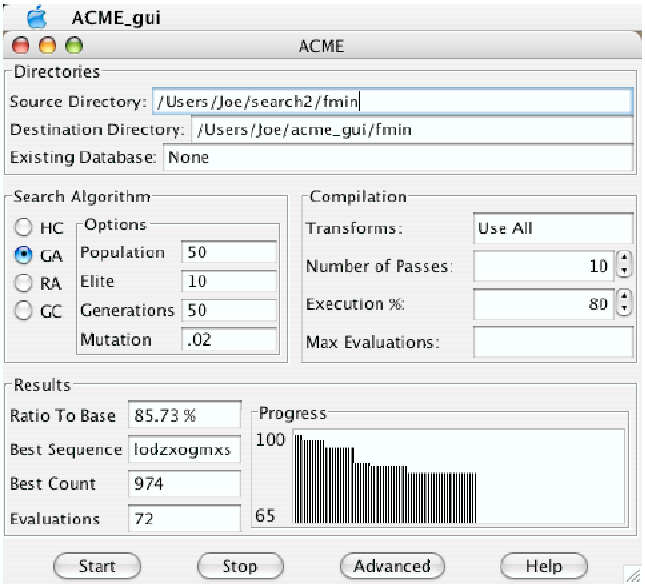

Figure 1. ACME Interface

3. ACME design

The design of ACME flows from our experience running tens of millions of compilations and includes both insights into the interface controls and engineering enhancements like virtual execution. Our goal has been to make the adaptive system both easier to use and more efficient.

3.1 Interface

An adaptive system should let the user ignore as many of the implementation-dependent details as possible. Obviously, some of the inputs must be entered by the user, but much of the control can either use default behavior or be hidden from the novice user. To that end, a GUI seems an obvious choice for an interface, since it can both show the user the necessary information and organize levels of information hierarchically, according to the skill or requirements of the user.

Figure 1 shows ACME's interface. The user must enter the name of the directory containing the code to be compiled, and ACME supplies default values for the rest of the parameters. The user can change any of the default parameters, including the search method, parameters for that method, *etc*. An advanced user may also wish to control such features as the set of optimizations, the random-number seed, and so on; these controls are found in the "Advanced" window, shown in Figure 2. Working through the interface, the following is a list of the notable features of ACME:

1. *Stop*. The stop button halts the search algorithm and returns the best result that ACME found. In conjunction with support for restarting a search from its last compilation (so the user doesn't

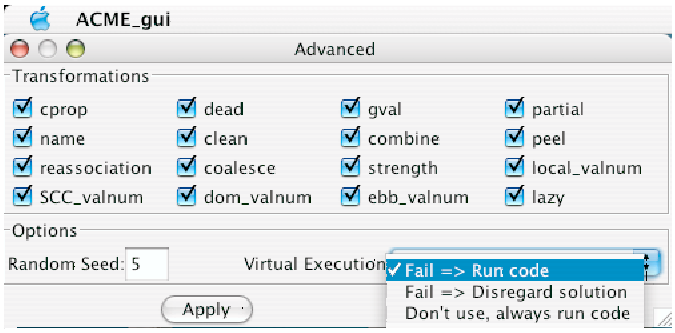

Figure 2. Advanced window

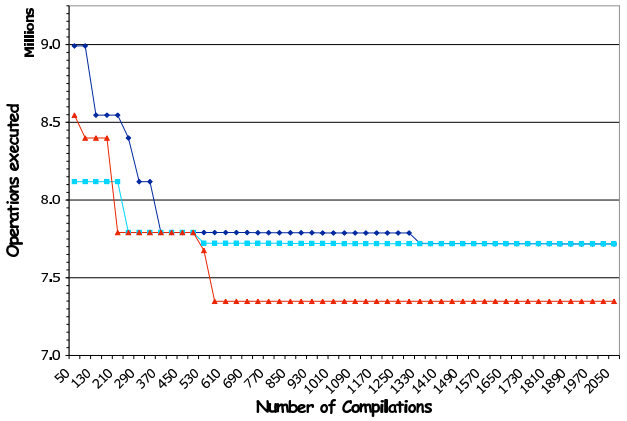

Figure 3. The progress of three successive runs of the genetic algorithm on `adpcm-decoder`

cause harm by accidentally pressing this button), the stop button may be the most important part of the user interface, because it gives manual control over a process that can run indefinitely long.

2. *Existing Database*. Whenever ACME runs, it stores the results of all compilation-string/execution-result pairs in a database. This database is stored in the "Destination Directory" along with any temporary files the compiler needs to create. It can be reused for subsequent invocations of ACME on the same piece of code. This enables the user, for example, to use his machine to compile overnight, stop the compilation in the morning, and then restart the compilation the next evening when he leaves. To take another example, the user can stop the compilation, run the code to see if it meets his needs, and then resume the compilation in the same place if it does not.
3. *Search Algorithms*. In [1], we show that different search algorithms have different cost/benefit tradeoffs. ACME currently supports four search algorithms: a greedy constructive algorithm, a genetic algorithm, a randomized impatient-descent algorithm (a hill climber), and random-probing search of the space. These give an expert user a high degree of flexibility, while the default hill-climber algorithm should give the novice user a good result quickly.
4. *Transforms*. ACME defaults to using all of the transformations available in the compiler, but the user may want to specify only a subset of these transformations, because they believe that the code may either not require a certain optimization pass (*e.g.*, if the code contains no loops, there is no reason to include loop-oriented transformations), or the pass may simply take too long (or be experimental or unreliable).
5. *Number of Passes*. This control allows the user to choose the length of the optimization sequence. We find that the default value of ten produces good results for our benchmarks (and serves our the experimental purposes), but an expert user may want to change this value; we have no quantitative data relating number of passes to quality of solution.
6. *Execution Percentage*. In benchmarks with many routines, it is often true that only a small number of the routines account for most of the work done during execution. The "execution percentage" variable tells ACME to start by profiling the code and recording the set of routines that account for the "percentage" of execution time as set by the user. Only this set of routines will be considered by the search algorithm; the infrequently executed routines can simply be ignored. If ACME is set to use virtual execution, the infrequently executed routines are simply ignored.
7. *Max Evaluations*. Some of the search routines (notably, the hill climber and the greedy constructor) will run for an unknown number of compilations. By setting this field, the user can bound the number of total compilations ACME performs, while leaving the field blank tells ACME to let the search algorithm run to completion.
8. *Progress Information*. Our experience in using our own system convinces us that feedback is critical. We start by compiling and executing the code using our standard optimization string. We then compare successive results during the search against this baseline. As better results are found, the "Best Sequence", "Best Counts" (the instruction-count measurement), and "Ratio to Base" fields are updated. The "Evaluations" field is a count of how many compilations and evaluations have occurred, to give the user a feel for the work being done. Lastly, the "Progress" graph shows the user how the results have improved over time.

   Experience shows that this data is particularly important, as exemplified by the graph in Figure 3. We performed an experiment in which we ran the genetic algorithm three times (*i.e.*, with three different random seeds) on the `adpcm-decoder` benchmark. The settings were generations of size 50, an elite set of 10 per generation, and 50 generations. These settings require 2050 compilations. In all three runs, we found the best answer by about the 650^{th} compilation, so that the rest of the time was wasted. This feature lets the user halt a search that has stopped making progress. The user can try again with a different seed or different search algorithm, or the user can choose to accept the solution. Because the database can be reused, redundancies across these restarts are avoided.
9. *Virtual Execution mode* (in the *Advanced* window). As we explain in Section 4.2, our current implementation of the virtual execution algorithm relies on an estimator. The estimator detects and reports cases in which it cannot give an accurate estimation of execution count. The default behavior in these difficult cases is to run the code to give accurate results. However, our experiments (shown in Figure 6) suggest that it may not hurt the solution quality to simply throw those compilations away.
10. *Random Seed* (in the *Advanced* window). All of the search algorithms rely to some extent on the generation of random numbers, and the generation of random numbers relies directly on the seed used to start the generator. In order to provide repeatability for our experiments, ACME defaults to using the same number as a seed to the search algorithms. The choice of random seed is transparent to the user who just wants to compile his code and then use it, but a researcher may want

`c`	*Sparse conditional constant propagation* [26]
`d`	*Dead code elimination* based on SSA-form [11, 10]
`g`	*Optimistic value numbering* [2]
`l`	*Partial redundancy elimination* [22]
`m`	*Renaming* builds the name space needed by the implementations of `l` and `z`. The compiler inserts it automatically before `l` or `z`.
`n`	*Useless control-flow elimination* [10]
`o`	*Peephole optimization* of logically adjacent operations [12]
`p`	*Peel* the first iteration of each innermost loop
`r`	*Algebraic reassociation* [5]
`s`	*Register-to-register copy coalescing* [6]
`t`	*Operator strength reduction* [10]
`u`	*Local value numbering* [10]
`v`	*Optimistic global value numbering* [23]
`x`	*Dominator-tree value numbering* [10]
`y`	*Extended-basic-block value numbering* [10]
`z`	*Lazy code motion* [18]

Table 1. Optimization passes included in ACME (the letters shown are used in the GUI to represent the transformations)

to have control over this value to be able to replicate a set of experiments or ensure that different runs produce different results.

3.2 Underlying design

The engine upon which ACME sits is the `iloc` compiler we have described in a number of other papers such as [4]. The list of optimizations included in the compiler is shown in Table 1. Each of the optimization passes is designed as a standalone Unix filter, which gives us the ability to easily reorder them arbitrarily. When a transformation sequence has been applied, we feed the code into our backend, which converts the `iloc` intermediate representation to `C`, which we compile using a native `C` compiler. This design allows us to instrument the code and run it as if it were on a virtual machine, independent of the actual architecture that we are using. This lets us run experiments on a variety of physical architectures and consolidate the results.

Further, measuring performance with instruction counts does not vary from run to run, in contrast to the timing measurements on our Unix systems. Timing measurements on a preemptive multitasking system vary enough to change the paths taken by the search methods, making it very difficult to get repeatability. Using instruction counts based on a virtual machine as our standard of measurement has proven to be controversial: the obvious objection is that the performance on modern architectures is heavily dependent on the behavior of the memory subsystem. However, none of the optimizations in our compiler specifically target memory performance in the same way as higher-level optimizations like loop transformations designed to improve data locality[21]. We have compared runtime measurements against instruction counts for the larger codes in our test suite using different optimization sequences, and they tend to correlate; that is, the instruction count measurements using different sequences tend to be separated by the same proportion as the runtime measurements of those sequences.

A second concern relates to the question of whether a single set of input data can correctly predict performance on arbitrary data. In our test suite of benchmarks, we have several codes with both training data that is used during adaptive compilation and test data that can be used to check the performance of the code. We tested for training bias by measuring the performance of the best version (obtained from the training data) on the testing data. In these experiments, we saw no systematic bias in the results – the performance improvement of the benchmark using the testing data was within 1 to 2% of the performance improvement of the training data.

The search algorithms are also implemented as C programs. They coordinate the running of the compiler and execution of the resultant code. They provide all of the bookkeeping, manage temporary files, and log results. To simplify the bookkeeping, test programs must maintain a strict design, too: they must reside in separate directories, and each program must have a configuration file containing some basic information such as the source-file names and input data to test the executable. The configuration files are easy to set up.

4. Eliminating the executions

In this section, we look at the theoretical and practical application of virtual execution. Virtual execution allows us to drastically reduce or completely eliminate the cost of the many executions that adaptive compilation normally requires.

4.1 Virtual execution

The concept of virtual execution relies on a simple premise: given optimizations which change only the code (but not the `CFG` – for example, loop-invariant code motion), two different versions of the same code produced from two different optimization sequences will always execute the same blocks for a given input. Virtual execution first counts each block's execution frequency with a profile of the unoptimized code. After this, any sequence of optimizations that adds, removes, or relocates instructions can be modeled by computing the sum over all the blocks of each block's frequency count multiplied by the number of instructions that end up in that block, and this measure should be precise[2].

The situation becomes more complicated when ACME includes optimization passes that *do* change the CFG. For example, consider loop peeling, an example of which is shown in Figure 4. This enabling optimization does nothing more than peel the first iteration of every loop in the program. In the figure, the clone of block A is denoted A′.

To update the execution-frequency counts for B and C, it is not correct to simply subtract one from B and C's frequency counts and set B' and C''s counts to one. Only one side of the conditional is taken on the first iteration of the loop, so the other side's block count should be set to zero, not one. Thus, we need something more from the initial profile than just the blocks' frequency counts if we are to handle optimization passes which modify the CFG. We need an actual path profile to handle this case – in fact, all of the CFG-changing passes in our compiler require this kind of information. While this can be expensive to gather, keep, and manipulate, the research in this area is extensive, and we feel confident that it is feasible [3, 13].

Of course, the optimization passes themselves must be augmented to maintain the path information as they make their respective changes to the CFG. While these changes should be fairly straightforward to implement, doing so may well be time consuming.

We should note one point. The underlying premise of virtual execution is that the order and frequency of execution of the basic blocks in a program need only be measured once, and any transformation which changes the CFG can likewise update the original counts to maintain accuracy. This premise relies on a restricted space of optimizations. For example, the premise may not hold when we extend our set of optimizations to include higher-level ones such as unroll-and-jam.

[2] This design has the additional advantage that it naturally takes into account varying instruction weights to allow us to consider different architectural features.

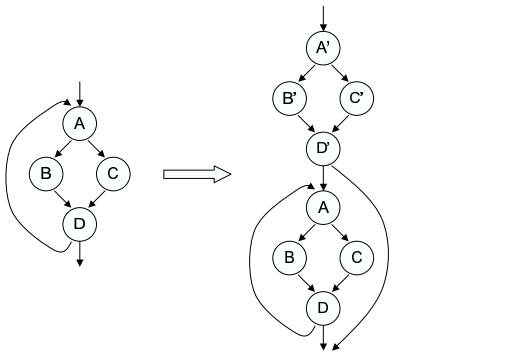

Figure 4. Example of loop peeling

4.2 Estimated virtual execution

The implementation of virtual execution requires updating any optimization pass which can change the CFG to simultaneously update the profiling data. This may well be an impediment to an existing compiler infrastructure, as it would necessitate considerable regression testing. As an alternative, we have developed a modified version of the virtual execution idea that we call *estimated virtual execution* (EVE). It is implemented as a separate pass inserted into the sequence after each invocation of a CFG-changing optimization pass.

The concept behind EVE is simple: rather than modify each pass that can change the CFG so that changes are accounted for as they occur, we build one pass that can compare the CFGs. Then, the compiler inserts this new pass after each CFG-changing transformation. The implementation works as follows. We run the unoptimized code and record, in each block, a block-identifying *tag* and an associated value showing the number of times the block executed. EVE looks at the tags to deduce the graph changes and update the execution counts of each block in the new CFG. It then assigns each block in the new CFG a unique tag with the newly updated count.

Consider again the loop-peeling example in Figure 4. After loop peeling, some blocks will be cloned and EVE will encounter the same tag in two blocks. The count that EVE will see in each of those two blocks will be the original count, and EVE must use other information to determine the new counts to associate with each block. Assuming that the nesting depth of the loop is one, we could deduce that the count of the block still in the loop is at most one less than the original count, and the count of the peeled block is at most one. We use the term "at most" carefully – only one side of the conditional in the peeled loop will execute, so the other side's count should be zero, but it may be impossible to determine this statically. The safe course is to assign both blocks in the peeled conditional a range from zero to one.

This becomes more complicated if the loop is nested within a second loop. When the inner loop is peeled, the number of times either side of the conditional in the peeled loop executes is impossible to know (if the outer loop runs ten times, it may be that every odd iteration takes the left path, every even iteration the right path, for example). Clearly, changes to the CFG can completely destroy our ability to update the block counts. As a result, EVE relies on a set of heuristics to deduce the counts of each block in the new CFG.

EVE uses techniques similar to static estimation to chart the changes in block execution frequency as the CFG changes, using the following assumptions:

1. If a basic block has not been duplicated or moved to a different nesting depth, its original frequency count remains unchanged.
2. If a basic block has been duplicated, the counts of the original and its clone(s) add up to the original frequency count.
3. If a new basic block has been inserted into the CFG, it does not affect other blocks' counts.

The estimator starts with a set of $(tag, count)$ pairs and a set of flow edges from the original CFG with their frequencies. It examines the updated CFG and predicts runtime block counts using the heuristics shown below. It is an iterative process, since determining a count for one block may make it possible to determine a count for other blocks (either directly or through the updated bounds). The heuristics use relationships of the blocks in the CFG such as dominance, post-dominance, and the successor and predecessor relationship and are as follows:

1. *Tags with zero count*: if any of the tags in a block has a zero count associated with it, the block receives zero count. There may be more than one tag if blocks have been merged.
2. *Trivial blocks*: if the loop nesting depth of block B has not changed, there are no other copies of this block with undetermined counts (no tag from B occurs anywhere else in the remaining blocks without counts), and all tags in B have the same count, then B receives that count.
3. *Compare upper and lower bounds*: compares the smallest and largest possible counts for each basic block. If block B's lower bound equals its upper bound, then the count for B is set to this value. This is a powerful heuristic because of the aggressive way that bounds are maintained.
4. *Compare the count assigned to a tag to the sum of upper bounds of all blocks containing this tag*: if the sum of upper bounds of blocks containing the same tag T is equal to the count assigned to T, each of the blocks receives a count equal to its upper bound.
5. *Compare the count assigned to a tag to the sum of lower bounds of all blocks containing this tag*: same as the heuristic for determining the upper bound (above), but using the lower bounds.
6. *Use counts for predecessors*: if the counts for all predecessors of block B (the set P made up of B's predecessors) have been determined and B is the only successor each of them has, then the count for B is the sum of counts of all predecessors. We implement an extension of this idea by allowing the blocks from P to have successors other than B, but require that their counts are already determined.
7. *Use counts for successors*: this is the same as using the counts from a block's predecessors (above), but applies to the successors.
8. *Use counts for edges*: if the count has been determined for block B, the counts for outgoing edges are available, and their sum is equal to B's count, then any successor S with B as its only predecessor receives the count of the edge $B \rightarrow S$. (At present, we do not implement the symmetric heuristic that inspects incoming edge counts.)
9. *Use edges with zero counts*: if a block A has only one predecessor B and the edge $B \rightarrow A$ has a zero count associated with it, then A receives zero for its count. On the other hand, if all edges leaving B, except for $B \rightarrow A$, have a zero count and the count for B is known, then A receives B's count.

If at some point no heuristic is applicable, but the derivation is incomplete, the estimator "guesses" the count for one of the blocks (for example, to maximize current total instruction count) and reapplies all heuristics. Sometimes this process leads to a reasonable solution. We save the state of the estimator before it performs its first guess: if it leads to contradictory counts or bounds we roll back to that point and try another value or pick another basic block.

The lower and upper bounds provide a range of possible runtime counts for each basic block. To be useful, they must be as accurate as possible. At the same time, the bounds may not be contradictory: a block's lower bound must be a non-negative value no larger than its upper bound; if the count for a block has already been determined, both lower and upper bounds must be equal to this count; the upper- and lower-bound values must agree with the counts and bounds for successors, predecessors, (post)dominators, and (post)dominated blocks.

Our implementation updates the bounds values after any of the blocks receives a count according to these observations:

- Already derived counts can be used as both lower and upper bounds for the corresponding block and propagated to other nodes in the CFG.
- A block cannot have a count larger than the count associated with any tag in it.
- A block will execute at least once if it dominates or postdominates all of its copies and its original count is greater than zero.
- Lower bounds can be used to improve upper bounds and vice versa.

To illustrate the last observation, consider a block B which has been duplicated by a transformation. If the upper bounds indicate that the maximum number of times all copies of B may execute is smaller than the original count for B (before the transformation), then B itself must account for at least the difference between the original count and the sum of upper bounds of the copies of B.

Similarly, the upper bound for B cannot be larger than the difference between its original count and the smallest possible number of times all other copies of B will execute (the sum of their lower bounds). While the sum of upper bounds may be larger than the original count, we always expect that the sum of the lower bounds is smaller than the block's original count. The lower bounds are updated for each predecessor (successor) of a block B to account for the difference between B's lower bound and maximum possible counts for all other predecessors (successors). Note that we cannot invert this observation to update upper bounds.

This third observation above is an instructive example of how EVE can fail, because the observation itself does not always hold true. On a few occasions, we discovered that two copies of a block (one copy dominating the other) would never execute, although the original count was greater than zero. A third copy of the same block existed but had been removed from the CFG at some point by the transformation, and EVE could not detect that.

Separate procedures implement the last observations, and we alternate between them until no more changes in the bounds are detected. Theoretically, the number of such alternations can be as large as the difference between upper and lower bound values; to limit such a circumstance, we interrupt this process after it repeats a fixed number of times.

Like static estimation, a certain amount of error is inevitable, because sometimes the technique either guesses wrong about the flow of control, or fails completely to understand how the original set of blocks are mapped into each new CFG. When the CFG changes in such a way that EVE can no longer be confident in the block-frequency counts, ACME can either run the code to obtain the precise counts, or it can fail to produce a result for that compilation. In general, the latter case just means that the search algorithm may have to run more trials. It can be more serious when ACME's estimate is inaccurate, because it may cause the search algorithm to make decisions based on erroneous data, meaning the solution found may not be as good as possible. Our experiments in the next section indicate that, despite the inaccuracies, ACME with EVE still achieves excellent results.

Code	*precise*	*within 1%*	*within 2%*	*within 3%*
fmin	34.5%	78.5%	89.2%	97.4%
zeroin	42.7%	68.5%	99.1%	100.0%
adpcm-c	39.1%	90.9%	96.1%	100.0%
adpcm-d	7.5%	100.0%	100.0%	100.0%
fpppp	24.2%	91.2%	95.5%	99.1%
nsieve	48.4%	81.6%	90.0%	94.7%
tomcatv	70.9%	93.3%	98.3%	99.8%
svd	50.8%	79.3%	88.7%	95.5%
ep	46.1%	66.9%	85.3%	99.1%
ft	80.7%	98.7%	99.3%	99.8%
is	64.8%	100.0%	100.0%	100.0%
mg	60.0%	93.7%	95.7%	97.4%
Mean	**47.5%**	**86.9%**	**94.8%**	**98.6%**

Table 2. The percentage of evaluations using estimated virtual execution falling close to the actual execution counts

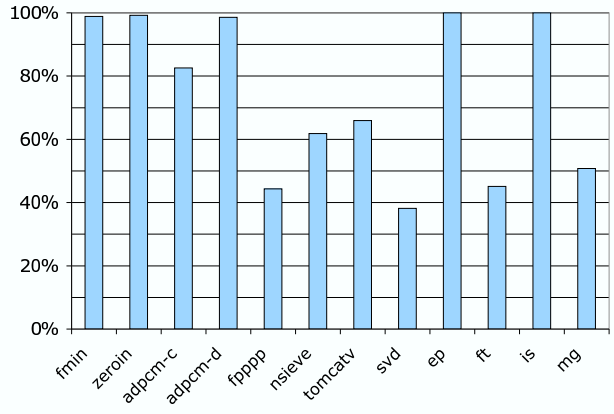

Figure 5. EVE success rate

5. Experimental Results

The techniques presented in this paper are designed to make the use of adaptive compilation systems practical. This section demonstrates that we can achieve results comparable to previous adaptive systems in substantially less time using ACME.

We performed all of our experiments on a Sun Fire V210 server. The server has two 1GHz processors, each with a 1MB cache, and 2GB of main memory. ACME was evaluated using twelve CPU-intensive applications taken from several different benchmark suites.

5.1 Evaluating Estimated Virtual Execution

EVE provides a fast alternative to executing code when evaluating optimization sequences in an adaptive system. However, since the instruction counts provided by this technique are imprecise, we first need to understand the degree of imprecision that is introduced and how this impacts the results achieved through adaptive compilation. It is also critical to understand the other aspect of this trade-off: the speedup gained through the use of EVE.

A basic measure of EVE's precision can be seen in Table 2. This table shows that the margin of error introduced by EVE is small in the vast majority of cases. As we said, we use instruction counts as our metric rather than machine timings precisely because we need both repeatability and accuracy in the adaptive searches. The data in Table 2 shows that EVE closely approximates actual instruction counts.

EVE is not always able to calculate an instruction count for a new version of a program. Complicated changes to the control

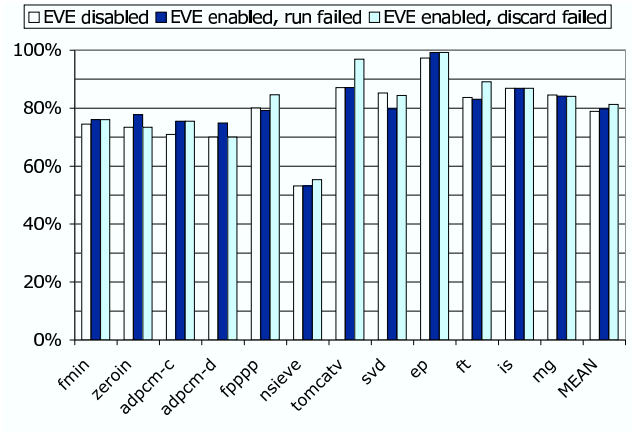

Figure 6. The quality of ACME's solution, measured against the default compilation sequence

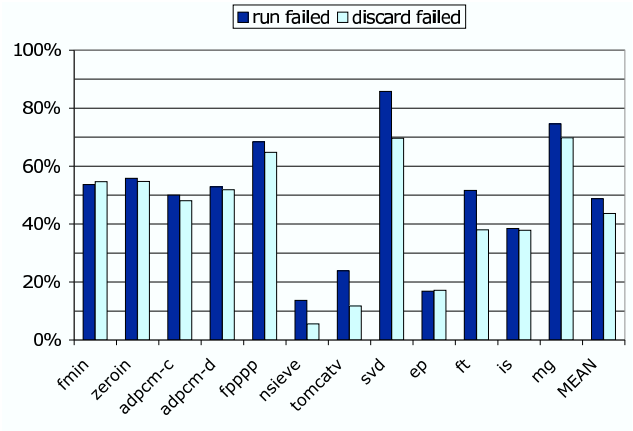

Figure 7. Running times of ACME with virtual execution enabled, normalized against full execution

flow of a program can cause EVE to fail. Figure 5 shows how often EVE successfully calculates an instruction count. There is a significant variation in the success rate across benchmarks. Several of the benchmarks pose no problem for EVE. However, for three of the benchmarks, EVE successfully understands the transformed programs less than half of the time.

To understand the value of EVE, we need to evaluate how it impacts the results we achieve through ACME. We used ACME to run a randomized hill climber limited to 500 total evaluations using either EVE or the traditional method of executing each program variation. We further divided the experiment into two cases: when EVE fails, we either just execute the code, or ignore the result and try another sequence. We chose a hill climber because it has been the most successful technique for finding good optimization sequences at this level of effort (generally 500 trials)[15]. At the same time, the hill climber's performance is the most sensitive to inaccurate estimations, which might mean that EVE would fail to find good solutions. The results, normalized against the performance of our standard optimization sequence, are shown in Figure 6. Using EVE in ACME provides performance close to the level of the standard adaptive compilation system, but without the overhead of repeatedly executing the code. EVE's inability to track the changes in some CFGs through some optimization sequences does not prevent ACME from ultimately finding a good sequence. The next section demonstrates how EVE speeds up adaptive compilation in ACME.

5.2 Running ACME

To determine the effects of EVE on the overall running time of the ACME compiler, we ran the hill-climbing search algorithm over our test suite with each of ACME's three execution modes as described in the previous section. Because each of the execution modes can cause the search algorithm to follow different paths, we set a cutoff of 100 evaluations for each target[3]. We normalized the times using EVE against the time without EVE. Because the search algorithm relies on a random-number generator, we ran this entire test multiple times using different values for the random seed to explore different areas of the optimization space and give a better picture of the benefits of EVE. The results, averaged across the multiple runs for each benchmark, are shown in Figure 7.

The timing results for these tests vary considerably for a variety of reasons. As we said above, the different execution modes will produce different searches by the hill-climbing algorithm. Some of the transformations take much longer to run than others, and these different searches employ different mixes of transformations, sometimes causing a wide variance in running time. For example, transformations based on data flow analysis such as partial-redundancy elimination may take significantly longer than a pass like our empty-basic-block cleaner[4]. Of course, if the code is executed, the variation in that execution time will impact overall running time, as well.

EVE cuts the adaptive compile time in half for many of our benchmarks and in several cases reduces the time to less than 30% of the time needed without it. Notable exceptions are `svd`, `fpppp`, and `mg`. As we showed in Figure 5, EVE fails a significant amount of time on these programs, because some of the routines in these programs have particularly complicated control flow. Interestingly, `ft` also presents problems for the EVE analysis but still shows a 50% decrease in overall compile time. The results are promising in this normalized view, but it is instructive to examine actual wall-clock time, as well.

The time required for ACME to do 100 evaluations of `nsieve` under the always-execute mode is approximately one hour. This is the highest observed for our set of benchmarks, followed by `fpppp` at 40 minutes, `tomcatv` at 18.5, and `ft` at 18 minutes. If we assume that it takes 500 evaluations for the hill climber to find an acceptable sequence, the time required for nsieve would be 5 hours if we execute the code for every sequence. With EVE engaged and discarding failed sequences, this wall time would drop to under 20 minutes. The next highest wall time is `fpppp` at just over 3.3 hours for 500 evaluations without EVE. This time would drop to under 2 hours, a less dramatic improvement, but still significant. For `tomcatv`, the total time would drop from approximately 1.5 hours to 11 minutes. For `ft`, 1.5 hours would drop to 35 minutes. While the degree of improvement varies, EVE certainly increases the range of programs that can be feasibly compiled with an adaptive system.

Some of the variations in the normalized results are due to the fact that virtual execution has no effect on the time it takes to apply the compilation sequence to the target program. For example, `nsieve` has a simple structure that allows the compilation sequence to complete quickly, so the ratio between transformation time and execution time is skewed toward execution time. As a result, virtual execution has a proportionally large benefit in this case. As we said in Section 2, other groups' work that targets reducing the number

[3] Ultimately, ACME ran a full 100 evaluations for each execution mode.

[4] In fact, partial redundancy elimination and lazy code motion both require the insertion of a renaming transformation that is not counted as a separate pass.

of compilations can be integrated with EVE to produce even better results.

6. Future work

As expert users of our own system, we would like to enhance ACME to handle hybrid search algorithms. For example, our experiments show that the genetic algorithm can be very useful for quickly finding a "good" solution, but it takes it a very long time after that to find a "better" solution. On the other hand, the hill climber will often find a "better" solution relatively quickly if it starts at a "good" solution. Thus, we would like an automatic way to let the genetic algorithm run for a short time, stop the compilation when we see it make significant progress (or even when its progress begins to level out, for example), and then restart the compilation with the hill climber, seeded with the good results obtained by the genetic algorithm[16].

Similarly, ACME could have an automatic mode, wherein it performs some relatively small number of random probes of the search space to empirically guess at the likelihood of many minima. It could then choose a search technique that might do well in that space, since the presence of fewer minima implies the necessity of using heavier weight search algorithms.

We currently have implemented a cutoff that allows the user to keep ACME from running forever by setting the maximum number of evaluations ACME can do. Some users may prefer a time limit rather than a compilation limit, so that, for example, ACME will search overnight for a compilation sequence, running as many searches as the time allows and delivering the best result first thing in the morning.

It may be that advanced users would prefer to see more detailed data of the search as it progresses. For example, the user might like to monitor the success rate of EVE. That data, combined with data showing how long a single execution takes, might guide the user as to what level of EVE to use – for short-running programs with complicated control flow, it may be better to disable EVE or execute the code when EVE fails.

Other suggestions from experienced users will undoubtedly change the face of ACME without changing its fundamental structure.

7. Conclusion

We have presented ACME, a system designed to support adaptive compilation. To overcome the steep learning curve of using a compiler of such complexity, we carefully designed a GUI that allows a novice user to easily use the system, at the same time allowing an advanced user to fine tune his compilation. Critical to its useability is the feedback the underlying compiler provides, making the compiler's progress visible to the user so that he can interrupt or redesign the adaptation if the need arises.

ACME includes four different search methods for the adaptation, and relies on estimated virtual execution to make compilation fast, sometimes an order of magnitude or more over actually executing the code for each compilation in the search.

We believe that ACME should serve as a model for future adaptive compilation systems.

8. Acknowledgements

There have been many people through the years who have spent considerable time and effort in helping to make the iloc compiler as useful a tool as it is. To those people go our heartfelt thanks.

References

[1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of LCTES 2004*, pages 231–239, June 2004.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of th 15^{th} POPL*, pages 1–11, San Diego, CA, USA, January 1988.

[3] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[4] Preston Briggs. The Massively Scalar Compiler Project. Appendix B of a nuweb document that forms part of the MSCP infrastructure. Available online as `www.cs.rice.edu/~keith/EAC/iloc.pdf`., July 1994.

[5] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of ACM SIGPLAN 94 PLDI*, pages 159–170, June 1994.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, January 1981.

[7] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Searching for compilation sequences. 2005. Submitted for publication.

[8] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of LCTES 1999*, pages 1–9, May 1999.

[9] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21^{st} century. In *Proceedings of the 2001 LACSI Symposium*. Los Alamos Computer Science Institute, Santa Fe, NM, October 2001.

[10] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann Publishers, 2003.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, October 1991.

[12] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM TOPLAS*, 2(2):191–202, April 1980.

[13] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Constrion*, pages 120–126, June 1982.

[14] Elana Granston and Anne Holler. Automatic recommendation of compiler options. In *Proceedings of the 4th Feedback Directed Optimization Workshop*, December 2001.

[15] Alexander Grosul. *Adaptive Ordering of Code Transformations in an Optimizing Compiler*. PhD thesis, Rice University, 2005.

[16] Jin-Kao Hao, Frédéric Lardeux, and Frédéric Saubion. A hybrid genetic algorithm for the satisfiability problem. In *Proceedings of the First International Workshop on Heuristics*, Beijing, China, July 2002.

[17] Toru Kisuki, Peter M.W. Knijnenburg, and Michael F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of PACT '00*, pages 237–248, October 2000.

[18] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 92 PLDI*, pages 224–234, July 1992.

[19] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Searches for effective optimization phase sequences. In *Proceedings of ACM SIGPLAN 2004 PLDI*, pages 171–182, May 2004.

[20] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whaley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Tools, and Compilers for Embedded Systems*, pages 12–23, June 2003.

[21] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, July 1996.

[22] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[23] L. Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.

[24] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 PLDI*, pages 77–90, May 2003.

[25] Spyridon Triantifyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the* 1^{st} *CGO*, March 2003.

[26] Mark Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM TOPLAS*, 13(2):181–210, April 1991.

[27] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM TOPLAS*, 19(6):1053–1084, November 1997.

[28] M. Zhao, B. Childers, and M.L. Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of LCTES 2003*, pages 1–11, June 2003.