ACME: Adaptive Compilation Made Efficient/Easy

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

ABSTRACT
Research over the past five years has shown significant performance improvements are possible using adaptive compilation. An adaptive compiler uses a compile-execute-analyze feedback loop to guide a series of compilations towards some performance goal, such as minimizing execution time.

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

We have developed a technique called virtual execution to decrease the time requirements for adaptive compilation. Virtual execution runs the program a single time and preserves information that allows us to accurately predict performance with different optimization sequences. This technology 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. It limits the amount of information that the user must provide to get started, by providing appropriate defaults. At the same time, it lets the user exert fine-grained control over the parameters that control the system. In particular, the user has direct and obvious control over the the maximum amount of time the compiler can spend, as well as the ability to choose the number of routines to be examined. (The tool uses profiling to identify the most-executed procedures.) The GUI provides an output screen so that the user can monitor the progress of the compilation.

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 problem of combining the correct set of optimizations for the panoply of input codes was a problem both recognized and generally ignored, because the complexity of reasoning about the interaction between transformations is dauntingly complex.

However, the exponential increase in processor speed has enabled experiments that use the computer itself to explore different combinations and permutations of optimization sequences. In short, these experiments have shown that the optimization sequences we believed were well conceived do not use the compiler to its greatest advantage[8, 9, 22]. This would be an interesting result if it stopped there, by identifying a maximally performing sequence of optimizations, but these experiments have also shown that different input codes benefit from remarkably different sequences. This secondary result argues strongly that a compiler that can change its behavior for each input will produce the best code.

There are two methods that a compiler can use to achieve this result. First, it can analyze the input code to detect features amenable to one or another transformation and optimize accordingly[24, 25]. From an academic standpoint, this would be the preferred method of operation, but it is currently beyond our capabilities: we do not yet know how to identify, in general, the salient characteristics of the input, nor do we have a vocabulary to describe the interaction between transformations which have markedly different effects on the code.

The second method for guiding a compiler's optimization sequence is the one we currently employ. Here, we use a feedback loop which starts by compiling the code with some sequence of optimizations, runs the code and produces a measurement, and then analyzes the result and instructs the compiler to recompile the code using a modified sequence of optimizations. We have experimented with a number of different analyses for guiding the compiler, including genetic algorithms, greedy algorithms, hill-climbing algorithms, and random probing. We have reduced the number of re-compilations from 20,000 in our initial experiments to
somewhat less than 500, using different methods of searching the space of sequences [8, 7]. We have shown consistent and significant improvements in code quality using these methods and have made arguments that the methods find optimization sequences that are within some small percentage of the optimal that can be found for a given set of transformations.

Even with the success of identifying efficient searching methods, the expense is still prohibitive for most users. Even 500 compilations and executions can take hours for our benchmarks. 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, but, because it (currently) enables four different searching algorithms with different sets of parameters, sixteen different optimizations, etc., it requires as many as fourteen different parameters for a single invocation of the compiler.

ACME addresses the performance problem by reducing the number of executions needed for adaptation to a single execution at the start of compilation. We use this to gather profiling data that we then use in the analysis phase of the adaptive compiler. The analyzer uses the profiling data to perform virtual execution (postulated upon in Section 4.1), a method of performance estimation based on instruction counts.

We address the problem of running a complex compiler with an easy-to-use interface that our experience shows provides correct levels of information to novices and more experienced users alike. It also provides the kind of feedback that all users would want to have.

We have used ACME to compare the running speed of the compilation with and without virtual execution, and have shown that virtual execution drastically reduces overall compilation time in our adaptive system.

2. RELATED WORK

Over the past several years there has been a great deal of research into adaptive compilation. Cooper, Schiedek, and Subramaniyan 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 compiler to produce better executables. Several researchers have continued to examine the problem of ordering optimizations [1, 9, 18, 17, 22]. 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 [15, 21, 25].

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 system with the most similar goals to ACME is VISTA.

Kullarni, et al.’s VISTA system is an interactive system which concentrates on reducing the compilation time, similar to ACME [17]. VISTA reduces 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. With virtual execution, ACME requires only a single execution of the code, regardless of the searching method used. VISTA also uses a number of techniques to weed out compilation sequences that probably will not change the code and so avoids unnecessary compilations. Other changes, such as a different library of transformations and ACME’s ability to run a variety of different searching methods, we believe are small. Indeed, we believe the techniques presented here are complementary to Kullarni et al.’s techniques rather than competitive.

3. ACME DESIGN

The design of ACME flows from our experience doing tens of millions of compilations and includes both engineering enhancements like virtual execution and insights into the interface controls. Our goal has been to make using the adaptive system less frustrating and more profitable.

3.1 Interface

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

In Figure 1, we show ACME’s interface. The user has only to enter the directory containing the code he wants to compile, and all of the rest of the parameters get default values, as shown. Alternatively, he can choose to change the searching method, searching parameters for each method, etc. An advanced user may also wish to control the optimizations, random seed, etc., and these controls are found in the “Advanced” window, as shown in Figure 3. The following is a list of the notable features of ACME:

1. **Stop** The stop button stops the searching algorithm and returns the best result ACME found. With other support such as restarting a search from its last compilation (so the user doesn’t hurt himself 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 temp files the compiler needs to create and 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 [7], we show that different search algorithms have different cost/benefit tradeoffs. ACME currently has four search algorithms: a greedy constructive algorithm, a genetic algorithm, an impatient random-descent algorithm (a hill climber), and a random-probing algorithm. These give an expert user a high degree of flexibility, while the default HC algorithm should give the novice user a good result for little effort.

4. Number of Passes This allows the user to control the length of the optimization sequence. We find that the default value of ten produces good results, but an expert user may want to change this value, trading compilation speed for quality – although, of course, there is no guarantee that a shorter string will not produce an excellent result, nor will a longer string guarantee that a good result will be found at all.

5. Code Percentage We found that on benchmarks with many routines, it is often true that only a small number of the routines account for the vast majority of the work done during execution. The “code percentage” variable tells ACME to start by profiling the code and recording the set of routines that account for the “percentage” of execution time set by the user. This set of routines will be considered by the searching algorithm, and the remaining routines’ object files will be stored for use during each compilation/execution phase.

6. Max Evaluations Some of the search routines (notably, the hill climber and the greedy constructor) will run an unknown number of compilations. By setting this field, the user can bound the number of total compilations ACME does, while leaving it blank tells ACME to let the search algorithm run to completion.

7. Virtual Exec pulldown menu As we explain in Section 4.2, our virtual execution algorithm relies on an estimator that can sometimes get confused. For completeness, the user may want to get an actual execution count when ACME cannot correctly estimate the count. Conversely, our experiments show that it may not hurt the solution quality to simply throw these compilations away.

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

9. Progress Information Finally, 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 2. 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 2550 compilations. In all three runs, we found the best answer by about the 650th compilation, so that the rest of the time was wasted. If the user can see that the improvement curve has flattened and the search is no longer finding better solutions, he can stop the search early and try again with a different seed, use the answer, etc.

3.2 Underlying design
The engine upon which ACME sits is the i386 compiler we have described in a number of other papers. Each of the optimization passes is designed as a standalone Unix filter, which gives us the ability to easily reorder them arbitrarily. Once we have finished optimizing, we feed the code
Sparse conditional constant propagation [23]
Dead code elimination based on SSA-form [11, 10]
Optimistic value numbering [2]
Partial redundancy elimination [19]
Renaming builds the name space needed by the implementations of 1 and z. The compiler inserts it automatically before 1 or z.
Useless control-flow elimination [10]
Peephole optimization of logically adjacent operations [12]
Peel the first iteration of each innermost loop
Algebraic reassociation [4]
Register-to-register copy coalescing [5]
Operator strength reduction [10]
Local value numbering [10]
Optimistic global value numbering [20]
Dominator tree value numbering [10]
Extended basic-block value numbering [10]
Lazy code motion [16]

Table 1: Optimization passes included in ACME

The optimizations included in the compiler are shown in Table 1.

We also have a number of C programs for the searching methods. They coordinate the running of the compiler and execution of the resultant code. They provide all of the bookkeeping concerning temporary files, logging results, etc. Because of some of this 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 files and input data to the executable. The configuration files are simple to set up, and really are just an idiosyncratic part of the system, rather than something intrinsic.

4. ELIMINATING THE EXECUTIONS
In this section, we look at the theoretical and practical application of virtual execution. This simple idea allows us to drastically reduce or completely eliminate 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), two different versions of the same code produced from two different optimization sequences will always execute the same blocks in the same sequence. We first count each block's execution frequency with a profile of the unoptimized code. After this, we can invoke any sequence of optimizations that add, remove, or relocate instructions and take an exact measure of the number of instructions the optimized code will execute 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.

This becomes more complicated when we include 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, we show the cloning of blocks by using a tick mark, and it is not correct to simply subtract one from \( B \) and \( C \)'s frequency count and set \( B' \) and \( C' \)'s counts to one.

Clearly, on the first iteration of the loop, only one side of the conditional is taken, so the other side’s block’s counts should be set to zero, not one, but which side?

Thus, we need something more from the initial profile than just a block’s frequency counts if we are to handle optimization passes which modify the CFG. We need an actual path profile to handle the case presented above – 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 this is feasible[3, 13]. Indeed, the depth of information is bounded by the number of passes employed: if the optimization sequence is \( t \) deep, we need only keep the path around \( t \) iterations of a loop (because, for example, the loop can be peeled only \( t \) times) and can summarize the remaining iterations.

Of course, the optimization passes themselves have to 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 that the premise relies on a restricted space of optimizations. For example, we are not confident that they extend to higher-level optimizations such as loop skewing.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>94.3%</td>
</tr>
<tr>
<td>zeroin</td>
<td>94.3%</td>
</tr>
<tr>
<td>adpcm-coder</td>
<td>96.0%</td>
</tr>
<tr>
<td>adpcm-decoder</td>
<td>96.6%</td>
</tr>
<tr>
<td>svd</td>
<td>78.9%</td>
</tr>
<tr>
<td>tomcat</td>
<td>99.1%</td>
</tr>
<tr>
<td>fp</td>
<td>97.6%</td>
</tr>
<tr>
<td>ep</td>
<td>99.0%</td>
</tr>
<tr>
<td>ft</td>
<td>96.4%</td>
</tr>
<tr>
<td>is</td>
<td>97.9%</td>
</tr>
<tr>
<td>mg</td>
<td>96.2%</td>
</tr>
</tbody>
</table>

Table 2: The percentage of evaluations using estimated virtual execution which are within three percent of actual execution counts

4.2 Estimated virtual execution

We wanted to measure the impact of virtual execution on ACME’s running time without major changes to the underlying compiler, so we designed a modified version of the virtual execution idea presented in the previous section that we call estimated virtual execution. Here, we insert into the sequence of optimizations a fix-up pass after each invocation of a CFG-changing optimization pass. This fix-up pass uses techniques similar to static estimation to chart the changes in block execution frequency as the CFG changes.

The estimator starts with a list of the original block names and their frequencies. It also maintains a list of the original set of flow edges and those frequencies. It then examines the updated CFG and tries to map its list against the current set of blocks, we make three assumptions about CFG changes:

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 sum of it and its clone’s frequencies must add up to its original frequency count.
3. If a new basic block has been inserted into the CFG, this does not affect other blocks’ counts.

Given these assumptions, we use a set of heuristics to derive frequency estimations for the set of blocks which fall outside the above restrictions. These heuristics use properties of the relationships of the blocks in the CFG such as dominance, post-dominance, the successor/predecessor relationship, etc.

Table 2 shows that this technique is sufficiently accurate to use in the system.

Like static estimation, this introduces a certain amount of error, 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 the ACME can no longer be sure of the block-frequency counts, it can either run the code and get the count that way, or it can just fail to enter a result for that string. In general, this latter case
just means that the searching algorithm may have to run more trials. It can be more serious when ACME's estimate is inaccurate, because it may cause the searching algorithm to make decisions based on erroneous data, meaning the solution found may be farther away from optimal than expected.

To discover the impact of the uncertainty and show that estimated virtual execution is viable, we evaluated our test suite of programs using both actual execution counts and results obtained by virtual execution. In Figure 5, we first show the success rate of estimated virtual execution — that is, the percentage of time when we ran a sequence of optimizations that produced code that ACME was unable to decipher. In five out of the twelve cases, ACME is able to successfully manage changes to the CFG. When the failure rate drops below sixty percent, it is usually because the code contains more complicated control flow than the algorithm can handle.\(^2\) Unfortunately, our solution quality does not fall by very much when this happens, almost certainly due to the fact that the search space does not have a high proportion of local minima — which forces us to use more expensive searching routines, rather than a small number of random probes. This mitigates somewhat the effect of failed estimations: the vast majority of optimization sequences in any search are far from a minima.\(^6\)

This, then, begs the question: even when estimated virtual execution manages to chart the changes to the CFG, how inaccurate will the estimate be? To evaluate this, we ran the hill climber with 50 restarts (HC-50) and the genetic algorithm with generation size of 50 and with 50 generations (GA-50). The results are shown in Figures 6 and 7. In all of these trials, we ran both versions of estimated virtual execution as suggested above: when virtual execution fails to identify and map CFG changes, the searching algorithm either: 1) runs the code and takes that count, or 2) simply ignores that compilation. These two approaches are labelled "ve" and "VE", respectively.

\(^2\)For example, a pass like the control-flow-cleaning pass often creates irreducible loops from reducible graphs.
In only one case did the inaccuracies of estimation cause us to get a significantly bad result. Indeed, in many cases, the different paths that the searches follow (because some estimations were different than some actual counts, and the searching algorithms made different decisions accordingly) actually arrived at slightly better results.

Remember that the purpose of this experiment was to show that our test harness is valid, not to suggest that using inaccurate measurements is somehow a desired search method. Having shown that estimated virtual execution performs very well against actual execution in terms of arriving at good solutions, in the next section, we will evaluate what this means to ACME's performance.

5. EXPERIMENTAL RESULTS

To be truly practical, adaptive compilation must be engineered to produce good results in a time frame that is tolerable. Our implementation of virtual execution greatly reduces the time that ACME must spend doing actual runs of the target program. We need to know how that effects the overall running time of the compiler. To determine this, we ran ACME and timed the results on our Apple Xserve, which uses a dual 1 GHz PowerPC G4 and 2 Gigabytes of memory. We used three modes of ACME for this test. The first mode did not use virtual execution at all - each candidate version of the target program was executed to determine its instruction count. The other two modes used virtual execution with the two options shown earlier, either running the target program when virtual execution fails or simply disregarding the compilation sequence. Then we normalized the times using virtual execution against the time without virtual execution. The results are shown in Figure 8.

Virtual execution cuts the overall compile time in half for all of our benchmarks except for svd, and in several cases ACME with virtual execution requires less than 30 benchmark is fpppp. The actual runtime of ACME without virtual execution for fpppp was over 3 hours. With virtual execution engaged and disregarding sequences for which it failed, the time dropped to about 31 minutes. Having ACME run fpppp when virtual execution failed increased the time to just under one hour. Either result could be obtained during one’s lunch break.

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, rsieve 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. Therefore, virtual execution has a large benefit in this case.

6. FUTURE WORK

As expert users of our own system, we would like to enhance ACME to handle hybrid searching algorithms. For example, our experiments show that the genetic algorithm can be very useful for quickly finding a “good” solution, but it takes a very long time after that to find a “better” solution. On the other hand, the hill climber will often find a “better” solution quickly if it starts at a “good” solution. Thus, we would like a 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.

Similarly, it has been suggested that 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 searching 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. It may be that users 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.

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

7. CONCLUSION

We have presented ACME, a compiler 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 usefulness 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 searching methods for the adaptation, and relies on 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 ACME compiler as useful as it is. To those people go our heartfelt thanks.

9. REFERENCES


