

An Adaptive Strategy for Inline Substitution

Keith D. Cooper¹, Timothy J. Harvey¹, and Todd Waterman²

¹Rice University Houston,
Texas, USA

`cooper,harv@rice.edu`

²Texas Instruments, Inc. Stafford,
Texas, USA

`twaterman@ti.com`

Abstract. *Inline substitution* is an optimization that replaces a procedure call with the body of the procedure that it calls. Inlining has the immediate benefit of reducing the overhead associated with the call, including register saves and restores, parameter evaluation, and activation record setup and teardown. It has secondary benefits that arise from providing greater context for global optimizations. These benefits can be offset by the effects of increased code size, and by deleterious interactions with other optimizations, such as register allocation.

The difficult aspect of inline substitution is choosing which calls to inline. Previous work has focused on static, one-size-fits-all heuristics. This paper presents a feedback-driven adaptive scheme that derives a program-specific inlining heuristic. The key contributions of this work are: (1) a novel parameterization scheme for the inliner that makes it susceptible to fine-grained external control, (2) a scheme for discretizing large integer parameter spaces, and (3) effective search techniques for the resulting search space. This work provides a proof of concept that can provide insight into the design of adaptive controllers for other optimizations with complex decision heuristics. Our goal in this work is *not* to exhibit the world's best inliner. Instead, we present evidence to suggest that a program-specific, adaptive scheme is needed to achieve the best results.

1 Introduction

Inline substitution is a simple transformation. It replaces a procedure call with a copy of the callee's body. The complex aspect of inline substitution is the decision process—the method by which the compiler decides which call sites to inline. The goal for inlining is to decrease the running time of the complete application. Unfortunately, the decision made at one call site affects the decisions at other call sites in subtle ways that can be hard to predict.

Naively, we would expect that inline substitution is always profitable. It eliminates operations to save and restore registers, to manipulate and manage activation records, and to evaluate parameters. On the other hand, studies have shown that inlining can increase application running time, due to effects that range from increased code size to decreased effectiveness of global optimization [7]. The impact at any single call site is multiplied by its execution frequency. Modern

programming practices encourage many calls to small procedures. An effective decision procedure must balance many competing effects.

Existing compilers typically attack this problem with a set of static heuristics. They aim to improve performance and avoid foreseeable pitfalls, such as excessive code growth. While these heuristics usually improve the speed of the compiled code, they leave a significant amount of improvement unrealized (See § 4).

The problem with static sets of heuristics is precisely that they are static; this paper shows evidence that different input programs need different strategies. For example, inlining small routines may produce radical improvement in one program but miss opportunities in another. The difference between a good inlining decision and a bad one often lies in low-level, idiosyncratic detail that is not obvious early in compilation—properties such as the demand for registers at the call site and in the callee, the execution frequency of the call, and the improved optimization that might accrue from knowledge about the actual parameters.

To address this problem, we designed and built an adaptive inliner. Many recent papers have explored aspects of adaptive optimization [25,28,3,21,15]. This work focuses on *program-specific optimization inside a single transformation*. That transformation has a huge and complex decision space; each decision changes the remainder of the problem. Our system includes a source-to-source inliner that runs quickly. It takes, as a command-line parameter, a closed-form description of an inlining heuristic and applies that heuristic to each call site in a predictable order. The design of that parameter scheme is critical to the system’s success. An adaptive controller manipulates the heuristic and measures the results to discover a good heuristic for the input program.

The next section explores the decision problem for inline substitution and introduces our inliner’s parameter scheme. Section 3 presents exploratory experiments that we performed to help us understand the decision spaces and to develop effective search techniques. The experimental results in Section 4 show that the adaptive inliner consistently produces faster executables than `gcc`’s inliner. Empirical evidence from our searches suggests why no set of static heuristics will work as well as the adaptive system across a broad range of input codes.

2 Designing a Parameter Scheme for Inlining

Inlining decisions are made with two levels of granularity. The compiler might decide to inline all calls to a particular procedure—for example, one whose body required fewer operations than the call would. On the other hand, the compiler might only inline a call if certain of the actual parameters have known constant values that permit optimization of the inlined body. Call-site specific decisions allow greater control, but make it harder to reason about the decision procedure.

To reason about the problem, assume that we have built a simplified call graph, $CG(N, E)$, with a distinct edge $\langle a, b \rangle$ for each call in a that invokes b and a map from each edge to a specific call site. To simplify matters, we will elide any backedges from the graph because the recursive cycles that they represent would add another layer of complication to the inliner.

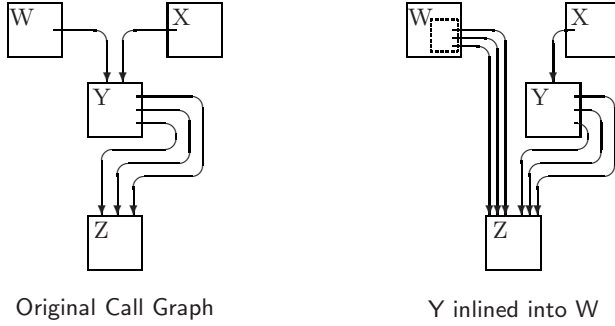


Fig. 1. An example call-graph

The problem of deciding which call sites to inline is hard, for several reasons. First, the impact of any particular inlining decision depends on the optimizer's effectiveness on the resulting code, which is hard to predict. Second, each decision to inline changes the call graph for subsequent decisions. To see this, consider the call graph in Figure 1.

The left side shows a call graph, where both W and X call Y , and Y calls Z from three distinct locations. Inlining Y into W produces the call graph on the right. Notice that the number of edges entering Z grows from three to six and the total edge count grows from five to seven. If the compiler, instead, inlines the calls from Y to Z , the total edge count shrinks from five to two.

If we view the decision problem as assigning a value, `inline` or `intact`, to each edge, the compiler initially faces a set of E decisions, one per edge and 2^E possible programs that it could consider and evaluate. Each time the system inlines an edge, it changes the graph. The decision can add edges; it can remove edges; it changes the program metrics for nodes and edges. Thus, the decision problem is solved in a complex and changing environment.

Finally, the decisions are interrelated and the order of decisions can affect the results. Consider, for example, a heuristic that inlines A into B if neither procedure has more than ten instructions. In Figure 1, if W and Y each have six instructions and Z has two, then inlining Y into W would prevent a subsequent decision to inline Z . On the other hand, inlining Z into Y first would not preclude inlining Y into W . Order has a direct effect on outcome; similar effects arise from the order of application of multiple heuristics.

The goal of this work was to construct a system that discovers good program-specific inlining heuristics. These program-specific heuristics are expressed in terms of concrete properties of the input program. The system measures those properties and then uses them in a feedback-driven tuning phase to construct a program-specific heuristic. The current set of program metrics are:

Statement count - **sc**: the number of C statements contained within a procedure.

It approximates procedure size. Many inlining heuristics try to mitigate object code growth by bounding the size of procedures that can be inlined.

Loop nesting depth - **lnd**: the number of loops in the caller that surround the call site. **lnd** proxies for execution frequency; calls in loops usually execute more often than calls outside of loops.

Static call count - **scc**: the number of distinct call sites that invoke the procedure in the original code. If **scc** is one, the call can be inlined with little or no code growth. If **scc** is small, the compiler might inline all of the calls and eliminate the original procedure.

Parameter count - **pc**: the number of formal parameters at the call. Because each parameter requires setup code, **pc** proxies for invocation cost. For small procedures, parameter setup costs can exceed the cost of inline execution [23].

Constant-parameter count - **cpc**: the number of constant-valued actual parameters at the call site. Constant-valued parameters may predict the benefits of optimizing the inlined code [4].

Calls in procedure - **clc**: the number of call sites inside the procedure that is a candidate for inlining. If **clc** is zero, the candidate is a leaf procedure. Such procedures are often small. Additionally, leaf procedures often account for a majority of total runtime.

Dynamic call count - **dcc**: the number of times that a call site executes during a profiling run of the program. Even minor savings at a site with high **dcc** can produce measurable improvement. **Dcc** captures profiling results and gives more accurate data on which to base inlining decisions [18,6].

These metrics were chosen to enable comparison with previous studies of inlining. *The set is neither complete nor definitive*, but it is broad enough to demonstrate the power of the parameter scheme. Adding or replacing metrics is easy.

The power of our design lies not in the specific metrics that it uses, but, rather, in the parameter scheme built on top of the metrics. The inliner takes a command-line parameter, the *condition string*, that specifies a complete heuristic. The condition string is a list of clauses in disjunctive normal form. Each clause is an inequality over the program metrics, literal constants, and arithmetic operators. For example, the clause **sc - clc < 10** specifies that any procedure comprised mostly of calls should be inlined. (To be precise, the expression evaluates to true if the number of statements that are not calls is fewer than ten.) Similarly, the condition string **lnd > 0 & sc < 100 | sc < 25** returns true for any call site in a loop where the callee has fewer than 100 statements and any call that lies outside loops where the callee has fewer than 25 statements.

To apply a condition string, the inliner evaluates the expression at every call site. It inlines each call site where the string evaluates to true. Call sites are considered in a postorder walk over the call graph, starting with leaves and working toward the root. When the system inlines a call site, it updates the program metrics to reflect the transformation.¹

The source-to-source inliner was built from Davidson and Holler’s INLINER tool [10] modified to accept ANSI C. It first builds the call graph and annotates

¹ The current metrics cannot express heuristics that rely on the inliner’s internal state: e.g., “*inline a call only if no other calls have been inlined into the caller.*” Adding a metric to account for this kind of data would be straightforward.

it with the various metrics described above. It reads the condition string from the command line. Next, it evaluates the condition string at each call, in a postorder walk of the call graph that ignores backedges. If the condition string is true, the tool inlines the call and updates both the graph and its annotations. The result is a transformed source program. In our tests, we compiled the programs with `gcc`; individual runs of the inliner took a small fraction of the time required to compile the code with `gcc`.²

3 Adaptive Search of the Parameter Space

The condition string model is critical to the success of our adaptive inliner. It is not, however, sufficient to guarantee success. For that, we need an adaptive framework that can efficiently find good condition strings for each input program.³ To guide our design efforts, we performed a series of preliminary experiments that improved our understanding of the search spaces encountered by the adaptive inliner. The experiments also provided insight into the relative importance of different program properties.

Our expressive parameter scheme creates immense search spaces. Thus, we could not exhaustively explore the spaces to improve our understanding. Instead, we conducted a series of one-, two-, and three-dimensional parameter sweeps to learn about the search space. Space constraints only permit us to discuss a small fraction of our preliminary experiments; Waterman's dissertation contains the full set of results [26].

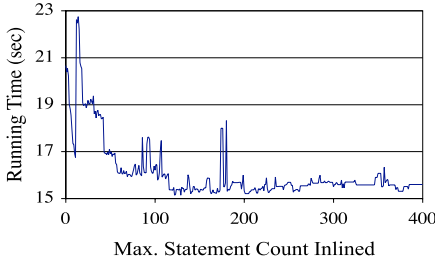
Figure 2 shows examples of the kinds of parameter sweeps that we performed. While the individual sweeps for different programs varied in detail, several general trends emerged. Specifically, the difference between best and worst performance was significant, and it required manipulation of several parameters to achieve the best performance.

The preliminary experiments provided two kinds of insights. First, they gave us an idea of which program properties measured by the inliner had the most impact on performance. Second, they gave us an idea of the shape of the search spaces. While the search spaces contain a variety of local minima and maxima, they appear significantly smoother than the search spaces seem in the phase-ordering problem [3]. The relative smoothness of the spaces for inlining makes them appear amenable to the use of hillclimbers.

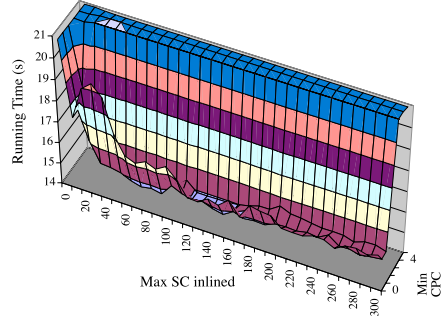
Pruning the Search Space. From the parameter sweeps, we learned that many parameter settings would achieve good results. Unfortunately, unrestricted

² We also experimented with Borland's C compiler for the PowerPC; both compile times and runtimes were longer than with `gcc`.

³ All experiments in this paper were run on a 1GHz, dual-processor G4 Macintosh running OS X Server. Each processor has a 256kB L2 cache and 2MB L3 cache. Benchmarks were inlined with the condition string being evaluated; the resulting code was then compiled using `gcc 3.3` with `-O3` enabled. Each experiment ran on an unloaded machine using just one processor. When needed, we disabled inlining in `gcc` with `-fno-inline-functions`.



1D Sweep of Statement Count

2D Sweep of Statement Count
and Constant Parameter Count**Fig. 2.** Parameter Sweeps with *vortex*

condition strings admit huge search spaces that are impractical to search. To limit the size of those search spaces, we adopted limits on the values of parameters and on the form of the condition string. These restrictions statically prune the search space. For example, the clause `sc > k` illustrates both problems. First, it admits an open-ended search for the unbounded parameter k . Second, the form of the clause limits the wrong aspect of the code; it places a lower-bound on the size of the inlined body rather than an upper bound. This clause will likely lead to large executables and little improvement. In contrast, the related clause `sc < k` has been used productively in several systems.

We first simplified the space of possible decisions by adopting a canonical form for condition strings. The canonical string is based on our own preliminary experiments and on insights from other studies. The current system limits conditions to the form:

$$\begin{aligned} & \text{sc} < A \mid \text{sc} < B \ \& \ \text{lnd} > 0 \mid \text{sc} < C \ \& \ \text{scc} = 1 \\ & \mid \text{clc} < D \mid \text{cpc} > E \ \& \ \text{sc} < F \mid \text{dcc} > G \end{aligned}$$

where A, B, C, D, E, F , and G are parameters chosen by the adaptive controller.

The first three clauses have been long been used as inlining heuristics. The first, `sc < A`, inlines any callee with fewer than A statements. Small procedures cause minimal code growth when inlined; each inlined call uses roughly the same calling sequence. Thus, the strategy of inlining tiny procedures seeks maximal benefit for minimal growth. The second, `sc < B & lnd > 0`, gives the controller a separate (presumably larger) bound for call sites inside loops because those call sites should execute more often. The third, `sc < C & scc = 1`, sets an independent bound for procedures called from just one site, since inlining these procedures causes no code growth. (The original copy of the callee can be discarded.)

The fourth clause, `clc < D`, inlines callees that contain fewer than D calls. For example, $D = 1$ inlines all leaves in the call graph. Raising the limit on `clc` produced strong results in our experiments, but has the potential to

increase code size rapidly. The fifth clause, $\text{cpc} > E \ \& \ \text{sc} < F$, tries to capture the potential benefits from inlining a call with constant-valued parameters. The importance of constant-valued parameters to inlining has long been recognized and studied [4]. We pair the constant-parameter count with an independent limit on callee statement count to limit code growth.⁴

The final clause, $\text{dcc} > G$, captures frequently executed call sites based on data from a profiling run [6,18]. Because the benefits of inlining are multiplied by execution frequency, inlining frequent calls is often a good strategy. This clause shows, again, the importance of a strategy to limit integer parameter values. The range of values for G can run from one to very large.

Bounding Integer Parameter Values. All of the parameter variables in the canonical string take on integer values. To limit further the search space, we must set reasonable bounds on the values for each variable—bounds that both delimit the search and avoid nonsensical values. These bounds must be program specific; for example, the upper limit on dcc should be small enough to admit some inlining of hot call sites and no larger than the maximum number of executions of any call site. Our system uses a variety of measures to establish bounds on the parameters. The goal is to limit the search without eliminating good solutions.

To bound the statement-count parameters, the adaptive controller performs a fast parameter sweep. It evaluates condition strings of the form “ $\text{sc} < X$ ”, with $X = 10$ initially. The sweep doubles X and reapplies the inliner until one of three conditions arises: memory constraints prevent compilation of the program; object code size has grown by a factor of ten; or the increase in X produces no change from the previous value. When one of these conditions occurs, the system uses the previous value of X as the maximum value for the statement-count parameter, with zero as the minimum. Other statement-count parameters are set to a multiple of this value, since they are constrained in other ways.

Bounds for the call count and constant-parameter-count variables are constants chosen from experience in our initial experiments. Call count shows good results with bounds up to three; however, $\text{clc} < 4$ produced exponential code growth in several of our benchmarks. Thus, the system limits clc to the range from one to three. With the constant-parameter variable, our tests have shown no significant benefit beyond a lower-bound of three. Thus, the system limits the lower bound on cpc to the range from zero to three.

The upper bound for dcc is set to its largest value observed in the profile run. To set a lower bound for dcc , the system uses a fast parameter sweep similar to the one used to select an upper bound for sc . The minimum dynamic call-count required for inlining is repeatedly reduced until one of the three conditions specified earlier occurs. Table 1 summarizes these procedures.

Discretizing Integer Parameter Ranges. While bounding the ranges of the integer parameters does reduce the search space, it still leaves impractically many points

⁴ While code size does not relate directly to speed, it does slow down compilation and, thus, the whole adaptive cycle. We found some huge executables that performed well, but the compile-time costs were large enough to call into question their practicality.

Table 1. Bounds for condition string parameters

<i>Parameter</i>	<i>Limits</i>	<i>Lower Bound</i>	<i>Upper Bound</i>
<i>A</i>	sc	0	<i>fast sweep on sc</i>
<i>B</i>	sc	0	10 * <i>fast sweep on sc</i>
<i>C</i>	sc	0	10 * <i>fast sweep on sc</i>
<i>D</i>	clc	0	3
<i>E</i>	cpc	1	3
<i>F</i>	sc	0	<i>fast sweep on c</i>
<i>G</i>	dcc	<i>fast sweep on dcc</i>	<i>max(dcc)</i>

to examine. Some of the variables, such as **dcc**, can have extremely large bounds. For example, profiling the **vortex** benchmark on our training data set produces a range for **dcc** that runs from 57 to more than 80 million. Obviously, 80 million discrete points is too many to search. To address this problem, we discretize each large range into a set of 21 points. With this crude discretization, the condition string still generates a search space of up to 49 million points.

For our strategy to be effective, the adaptive controller must establish a good distribution of the twenty-one search points throughout the range of each discretized variable. Our first experiments divided the search points linearly across the range. We quickly realized that many parameters have extremely large ranges, but that the interesting values tend to be grouped together at lower values. We experimented with a quadratic distribution, but it succumbs to the opposite problem: points are grouped too closely together at the low end, and too sparsely at the high end. Our best results came with a hybrid of these two, a quadratic distribution with a linear coefficient: $value = c_1x^2 + c_2x$. Currently, the system sets c_2 to five, a value chosen after some tuning. The quadratic coefficient c_1 is program and parameter specific; it is chosen to generate the desired minimum and maximum values. Table 2 shows how these different approaches divide the parameter space for **sc** in **vortex**.

The system uses a different scheme to distribute the search points for **dcc**. Because the controller has a **dcc** value for each procedure in the program, it can distribute **dcc** values based on actual data. Thus, it uses a formula based

Table 2. Division of the **sc** parameter for **bzip2** using different distributions

<i>Ordinal</i>	0	1	2	3	4	5	6	7	8	9	10
<i>Linear</i>	0	256	512	768	1024	1280	1536	1792	2048	2304	2560
<i>Quadratic</i>	0	13	51	115	205	320	461	627	819	1037	1280
<i>Hybrid</i>	0	17	58	123	212	325	462	623	808	1017	1250

<i>Ordinal</i>	11	12	13	14	15	16	17	18	19	20
<i>Linear</i>	2816	3072	3328	3584	3840	4096	4352	4608	4864	5120
<i>Quadratic</i>	1549	1843	2163	2509	2880	3277	3699	4147	4621	5120
<i>Hybrid</i>	1507	1788	2093	2422	2775	3152	3553	3978	4427	4900

on percentiles in the measured range. If the program has 1000 call sites with an execution frequency between the previously determined maximum and minimum, the first search point would be the `dcc` of the most executed call site. The second search point would be the dynamic call count of the 50th most executed call site ($\frac{1000 \text{ points}}{20 \text{ intervals}} = 50 \frac{\text{points}}{\text{interval}}$). In our experience, this distribution works well for `dcc`. We cannot use this approach with `sc` because the statement counts of a procedure change during the process of inlining. (The bottom-up order of inlining decisions means that `dcc` cannot change until after it has been used.)

Searching the Spaces. Given the canonical condition string and a set of bounds on the parameters, the question remains: how should we search this space? We elected to use a hill-climbing algorithm. As its first step, the hillclimber selects a point in the search space at random and evaluates it. Next, it evaluates the current point’s neighbors, in random order. If it finds a neighbor with better results, it shifts the current focus to that point and explores that point’s neighbors. This process continues until no better neighbor can be found, indicating a local minimum. Our experimental results suggest an approach that makes several descents from distinct, randomly-chosen starting points.

To implement the hillclimber, we need a concrete notion of “neighbor.” The canonical condition string has seven variables that the controller can vary. We define the immediate neighbors of a point to be those arrived at by increasing or decreasing a single parameter. Thus, each point in the space defined by the canonical string has 14 potential neighbors.

The hillclimber takes the first downward step that it finds, rather than evaluating all of the current neighbors and taking the best step. Thus, it makes a *random descent* rather than a *steepest descent*. Random descent takes, on average, fewer evaluations per step. In the relatively smooth spaces that we have observed, it is more cost effective to make additional random descents than to perform fewer of the more expensive steepest descents.

Constraining the Start Point. The pruned search space still contains points that make poor starting points for the hillclimber. Some points produce such large transformed programs that they either fail to compile or compile so slowly that as to make the search impractical. (The hillclimber regards failure to compile as producing an infinite execution time.) Unfortunately, such points often have neighbors with the same problem. Thus, starting the search at such a point often produces no progress or painfully slow progress.

To avoid starting a search at such a point, we added a further constraint for start points. A start point must satisfy the condition: $A^2 + B^2 + C^2 + D^2 + E^2 + F^2 + G^2 \leq 400$, for the variables A through G in the canonical string. This constraint allows a single parameter at its maximum value, or several parameters with large values, but eliminates the unsuitable start points where too many parameters all have large initial values. The restriction only applies to a start point; the subsequent search can move outside these bounds.

Table 3. Improvements From Adaptive Inlining

<i>Method</i>	vortex			bzip2			mcf			parser		
	<i>Time</i>	<i>%</i>	<i>Dev</i>	<i>Time</i>	<i>%</i>	<i>Dev</i>	<i>Time</i>	<i>%</i>	<i>Dev</i>	<i>Time</i>	<i>%</i>	<i>Dev</i>
<i>gcc no inl'g</i>	20.95	100		73.45	100		48.46	100		15.81	100	
<i>gcc inl'g</i>	17.97	86		71.89	98		46.94	97		13.30	84	
<i>1 descent</i>	15.21	72	0.51	71.40	97	1.84	47.09	97	0.42	12.41	79	0.13
<i>Best of 5</i>	14.68	72	0.29	68.91	97	1.69	46.62	97	0.37	12.29	79	0.10
<i>Best of 10</i>	14.52	69	0.22	68.15	93	1.55	46.36	96	0.07	12.14	77	0.10
<i>Best of 20</i>	14.39	69	0.05	67.67	92	1.56	46.30	96	0.00	12.14	77	0.07

4 Experimental Results

To evaluate the adaptive inliner, we ran a set of experiments to measure the effectiveness of the inlining strategy and the effectiveness of the search strategy. In each experiment, we used the adaptive inliner to create a transformed source program, which we then compiled with `gcc`. We compared the results of our system against the original source code and against the results of the `gcc` inliner. We performed the experiments using the same computer setup that we described in Section 3. Due to space limitations, we will focus our discussion on four benchmark codes, **vortex**, **bzip2**, **mcf** and **parser**. Again, Waterman’s dissertation provides more detailed results[26].

To assess the potential impact of the adaptive inliner, we ran one hundred descents of the hillclimber on each benchmark and recorded both the running time and the condition string for each run. Using the hundred descents, we computed the average performance from a single descent (average over 100 descents), a best-of-5 run (average of 20 best-of-5 runs), a best-of-10 run (average of 10 best-of-10 runs), and a best-of-20 run (average of five best-of-20 runs). Table 3 shows these results, along with the running time for the code with no inlining and with `gcc 3.3`’s inliner.⁵ Improvements range from 4% to 31%.

On these four benchmarks, the `gcc` inliner always produces faster code than the original. (We saw one benchmark, **gzip**, where `gcc`’s inliner produced a slowdown.) The adaptive inliner, in general, outperforms `gcc`’s inliner. A single descent usually beats `gcc`; on **mcf**, the average single descent produced a slightly slower version than `gcc`’s inliner. As a trend, more descents produce better results, although the returns appear to diminish beyond ten descents. The column labelled *Dev* shows the standard deviation in running time across the multiple “best-of-*x*” runs. While the *average* result from a single descent is close to that of a best-of-10 run, any single descent may well be far away from a better solution. Such variability is a natural result of using randomized greedy algorithms.

The improved performance and consistency from multiple descents strongly encourage such an approach. Of course, increasing the number of descents

⁵ To maximize the performance of `gcc`’s inliner, we moved all the source code for each application into a single file.

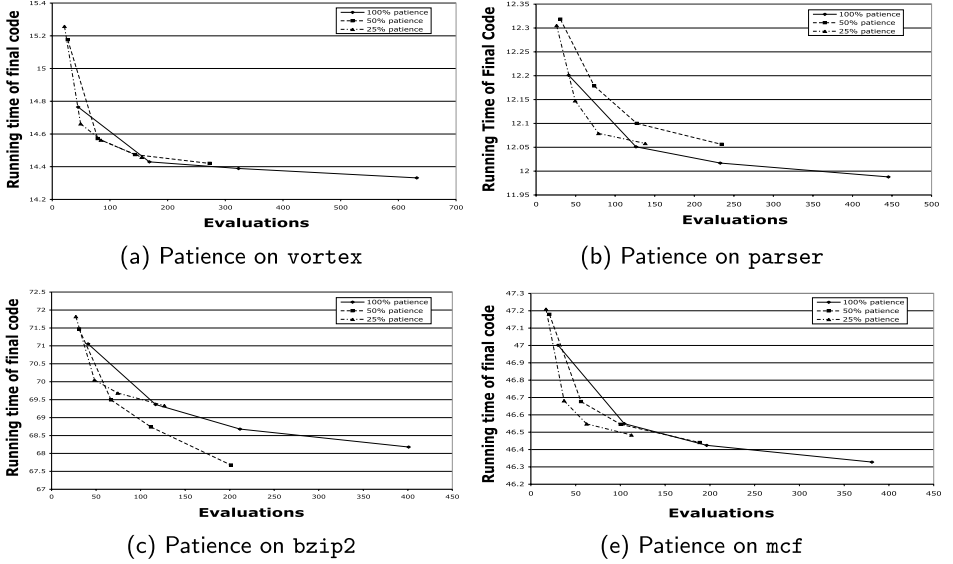


Fig. 3. Effects of patience on benchmark programs

increases the overall cost of finding a good heuristic. The hillclimber, at five to ten descents, seems to represent a good point on the cost/benefit curve. (Remember, the developer need not run the adaptive system on every compile. Because it returns the condition string, subsequent compiles can simply use that program-specific heuristic. The adaptive step need only be redone when program properties change in a significant way.)

To explore the tradeoff between the number of evaluations and the speed of the resulting executable, we conducted experiments with an *impatient hillclimber* [3]. A *patient hillclimber* examines each neighbor of the current point before it declares the current point to be a local minimum. An impatient hillclimber limits the number of neighbors that it will examine before deciding the current point is a local minimum. That limit, the *patience factor*, is expressed as a percentage of the neighbors. The implementation of an impatient hillclimber must select the neighbors in random order.

Because an impatient hillclimber can terminate its search prematurely, it will miss some good solutions. However, the lower cost per descent of an impatient hillclimber allows the compiler to perform more descents—to restart the search from another randomly chosen start point. We have shown elsewhere that impatient hillclimbers are effective in spaces that have a sufficient number of “good” solutions [3,15]. These experiments suggest that the search spaces in which the adaptive inliner operates have that property.

The graphs in Figure 3 show the effects of patience on both effort and solution quality for the adaptive inliner. For each benchmark, the graph shows behavior of a patient hillclimber (100% patience) and two impatient hillclimbers. We

Table 4. Frequency with which each neighbor was chosen as the downward step

Step	vortex	parser	bzip2	mcf
sc Increased	7.88%	11.17%	15.74%	9.30%
sc Decreased	9.07%	19.68%	21.30%	22.10%
Loop sc Increased	8.11%	10.64%	0.93%	1.16%
Loop sc Decreased	8.35%	8.51%	1.85%	3.49%
scc sc Increased	13.60%	10.11%	23.15%	20.93%
scc sc Decreased	5.25%	8.51%	12.04%	34.88%
clc Increased	3.82%	4.26%	8.33%	2.33%
clc Decreased	3.82%	2.12%	2.78%	2.33%
cpc Increased	3.58%	5.32%	2.78%	2.33%
cpc Decreased	4.06%	1.59%	4.63%	1.16%
cpc sc Increased	6.44%	3.19%	0.00%	0.00%
cpc sc Decreased	3.34%	0.53%	0.00%	0.00%
dcc Increased	18.85%	4.26%	1.85%	0.00%
dcc Decreased	3.82%	10.11%	4.63%	0.00%

examine 50% and 25% patience since they provide sufficient savings to make limited patience worthwhile. Lower values would examine just one or two neighbors for each point. For these search spaces, those numbers are just too small.

The graphs show average results taken over 100 restarts. The graphs show a common and expected theme: if the system is limited to a fixed number of evaluations, an impatient hillclimber produces better results than a patient hillclimber. Multiple descents are more important to protect against a bad start point than is a thorough exploration of the neighborhood. If the number of evaluations is not a concern, multiple patient descents usually provide better results.

The results for **bzip2** are an anomaly that demonstrates the noise involved in random greedy algorithms. For **bzip2**, we obtained better results with 50% patience than with 100% patience. We attribute the difference between the runs to the fact that they both used randomly selected starting points. Clearly, the runs at 50% patience found more effective starting points.

A final experiment that provides insight into the search space is shown in Table 4. We examined the fraction of the time that each neighbor was chosen as the downward step in the hillclimber, across the various benchmarks. Originally, we were looking for bias on which we could capitalize. The experiment showed little bias; in fact, every possible change occurs a significant percentage of the time for at least some benchmarks. This experiment produced two important conclusions. First, each parameter in the condition string plays a role in some set of decisions. This fact shows that the inliner should examine a variety of program properties. The condition string and the program metrics capture important aspects of the problem and its search spaces.

Second, the most frequently chosen parameters vary by benchmark. Thus, we should not bias the hillclimber, since trends do not carry across benchmarks. In itself, this observation is important. If the same parameters were adjusted in the same ratio across different benchmarks, then it would suggest the possibility

of a universal solution to finding sets of inlining decisions. However, the wide variation in winning parameters across the benchmarks strongly supports the notion that different heuristics are needed for different programs. It reinforces our belief that adaptive inlining should outperform any static heuristic.

Taken together, these results demonstrate the efficacy of adaptive inlining. The adaptive inliner consistently outperforms the traditional inlining approach embodied in `gcc`; it also outperformed no inlining. In the one case where `gcc`'s inliner degraded performance (`gzip`), the adaptive system discovered a heuristic that ran 15% faster than the original code. The quality of results varies based on the number of evaluations provided to the adaptive system, but good results can be obtained with a small number of evaluations using limited patience. Finally, examining the neighbors chosen by the hill climber's descent demonstrates that different sets of inlining decisions are appropriate for different programs and that an adaptive approach is necessary to capitalize fully on the potential of inlining.

5 Related Work

Our work touches upon two separate areas of compiler technology, adaptive control of optimization and inline substitution. Both have an extensive background in the literature. Space constraints prevent us from providing a complete related work section.

Adaptive control of optimization has been widely explored in recent years. Work has included profile-based optimization [6,13] and self-tuning libraries [27]. Several groups have looked at the problem of selecting and ordering optimizations [8,28,25,3,21]; those papers have used a variety of adaptive control mechanisms ranging from genetic algorithms [8,21] to feedback-driven search [25,3] to model-driven algorithms [28]. Other authors have looked at selecting command-line options for the compiler [14] and deriving parameters and heuristics for specific optimizations [19,24,9]. None of these studies examined flexible, expressive parameter schemes similar to ours.

Inlining has a long history in the literature, dating back to the early 1970's [2,20] [1,16,17,22]. Many authors have studied the decision problem for inlining [4,10,7] [11,18,12,29]. The closest work to our own is by Cavazos and O'Boyle [5]. They used a genetic algorithm to tune the inlining heuristic in the Jikes RVM for new platforms. Their system derived a platform-specific, program-independent inlining heuristic; it was run once per platform as part of the process of porting the RVM. In contrast, our system uses impatient search to find reusable but program-specific inlining heuristics.

6 Conclusions

This paper presents the design of an adaptive inliner and results of an experimental validation of that tool. Our tool constructs a program-specific heuristic to make inlining decisions and applies that heuristic to produce a transformed version of the source compiler. Individual decisions are made by applying the

heuristic on a call-site by call-site basis; the heuristic uses measured properties of the call site, caller, and callee. Our system measures seven specific program properties, but the scheme is easily extended to include new metrics.

To validate our ideas and our design, we compared the performance of the original programs, of those programs as optimized by `gcc`'s inliner, and of those programs optimized in our system. All the versions were compiled with `gcc`, run, and measured. The results show that the adaptive system finds program-specific inlining heuristics that consistently outperform `gcc`'s inliner and the original programs. Careful analysis of the adaptive system's behavior suggests that no single heuristic can achieve equivalent results across diverse programs.

Design and development of this system led to many insights. Two are particularly important. The parameterization used to express the decision heuristic lets the adaptive controller express and explore a huge decision space and allows it sufficiently precise control to obtain good results. In our experience, compilers and optimizations do not provide an interface that allows effective external control; parameter schemes similar to ours would be a substantial improvement. The techniques that we developed to deal with discretizing large integer spaces and to search them efficiently may help others as they develop adaptive controllers for other complex optimizations. Our solutions should provide a starting point for exploring those spaces and building new search algorithms.

References

1. Allen, F., Carter, J., Fabri, J., Ferrante, J., Harrison, W., Loewner, P., Trevillyan, L.: The experimental compiling system. *IBM Journal of Research and Development* 24(6), 695–715 (1980)
2. Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In: Rustin, J. (ed.) *Design and Optimization of a Compiler*, pp. 1–30. Prentice-Hall, Englewood Cliffs (1972)
3. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, June 2004, pp. 231–239 (2004)
4. Ball, J.E.: Predicting the effects of optimization on a procedure body. In: *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, August 1979, pp. 214–220 (1979)
5. Cavazos, J., O'Boyle, M.F.P.: Automatic tuning of inlining heuristics. In: *Proceedings of the 2005 ACM IEEE Conference on Supercomputing (SC 2005)* (November 2005)
6. Chang, P.P., Mahlke, S.A., Hwu, W.W.: Using profile information to assist classic code optimizations. *Software—Practice and Experience* 21(12), 1301–1321 (1991)
7. Cooper, K.D., Hall, M.W., Torczon, L.: An experiment with inline substitution. *Software—Practice and Experience* 21(6), 581–601 (1991)
8. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999, pp. 1–9 (1999)

9. Cooper, K.D., Waterman, T.: Investigating adaptive compilation using the MIP-Spro compiler. In: Proceedings of the 2003 LACSI Symposium, October 2003, Los Alamos Computer Science Institute, Santa Fe, NM (2003)
10. Davidson, J.W., Holler, A.M.: A study of a C function inliner. *Software—Practice and Experience* 18(8), 775–790 (1988)
11. Davidson, J.W., Holler, A.M.: Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering* 18(2), 89–102 (1992)
12. Dean, J., Chambers, C.: Towards better inlining decisions using inlining trials. In: Proceedings of the 1994 ACM Conference on LISP and Functional Programming, June 1994, pp. 273–282 (1994)
13. Fisher, J.A.: Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* C-30(7), 478–490 (1981)
14. Granston, E., Holler, A.: Automatic recommendation of compiler options. In: Proceedings of the 4th Feedback Directed Optimization Workshop (December 2001)
15. Grosul, A.: Adaptive Ordering of Code Transformations in an Optimizing Compiler. PhD thesis, Rice University (2005)
16. Harrison, W.: A new strategy for code generation - the general purpose optimizing compiler. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, January 1977, pp. 29–37 (1977)
17. Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier North-Holland, New York (1977)
18. Hwu, W.W., Chang, P.P.: Inline function expansion for compiling C programs. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, June 1989, pp. 246–257 (1989)
19. Kisuki, T., Knijnenburg, P., O’Boyle, M.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, October 2000, pp. 237–248 (2000)
20. Knuth, D.E.: An empirical study of FORTRAN programs. *Software—Practice and Experience* 1(2), 105–133 (1971)
21. Kulkarni, P.A., Hines, S.R., Whalley, D.B., Hiser, J.D., Davidson, J.W., Jones, D.L.: Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.* 2(2), 165–198 (2005)
22. Scheifler, R.W.: An analysis of inline substitution for a structured programming language. *Communications of the ACM* 20(9), 647–654 (1977)
23. Serrano, M.: Inline expansion: *when* and *how*? In: Serrano, M. (ed.) Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs, September 1997, pp. 143–147 (1997)
24. Stephenson, M., Amarasinghe, S., Martin, M., O’Reilly, U.-M.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (June 2003)
25. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback directed and runtime-optimization, March 2003, pp. 204–215 (2003)
26. Waterman, T.: Adaptive Compilation and Inlining. PhD thesis, Rice Univ. (2005)

27. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. *Parallel Computing* 27(1–2), 3–25 (2001)
28. Zhao, M., Childers, B., Soffa, M.L.: Predicting the impact of optimizations for embedded systems. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Tools, and Compilers for Embedded Systems*, June 2003, pp. 1–11 (2003)
29. Zhao, P., Amaral, J.N.: To inline or not to inline? enhanced inlining decisions. In: *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, October 2003, pp. 405–419 (2003)