Aggressive Inlining

Andrew Ayers Robert Gottlieb Richard Schooler

Hewlett-Packard Massachusetts Language Laboratory
300 Apollo Drive
Chelmsford, MA 01824
e-mail: {ayers,gottlieb,schooler}@ch.hp.com

Abstract

Existing research understates the benefits that can be obtained from inlining and cloning, especially when guided by profile information. Our implementation of inlining and cloning yields excellent results on average and very rarely lowers performance. We believe our good results can be explained by a number of factors: inlining at the intermediate-code level removes most technical restrictions on what can be inlined; the ability to inline across files and incorporate profile information enables us to choose better inline candidates; a high-quality back end can exploit the scheduling and register allocation opportunities presented by larger subroutines; an aggressive processor architecture benefits from more predictable branch behavior; and a large instruction cache mitigates the impact of code expansion. We describe the often dramatic impact of our inlining and cloning on performance: for example, the implementations of our inlining and cloning algorithms in the HP-UX 10.20 compilers boost SPECint95 performance on a PA8000-based workstation by a factor of 1.32.

1 Introduction

Procedure boundaries have traditionally delimited the scope of a compiler's optimization capabilities. Indeed, it is no accident that optimizations within a procedural scope are termed *global* optimizations. But as an optimizer's scope is limited, so is its power, and several techniques have been developed to extend optimizations to larger scopes.

One such technique is *inlining*: direct incorporation of the code for a subroutine call into the calling procedure. After inlining, optimizations blocked or hindered by the procedure call boundary can be applied straightforwardly to the combined code of caller and callee with little or no loss of precision. As a side benefit, the run time cost of a procedure call is also eliminated. Another common technique for exploiting interprocedural information is *cloning*: the duplication of a callee so that its body may be specialized for the circumstances existing at a particular call site or set of call sites.

Inlining is often considered to be a brute-force approach to interprocedural optimization. Since many global optimizations are not linear time or linear space, and since instruction caches are of fixed capacity, the code expansion caused by inlining is cause for some concern. Interprocedural analysis is usually proposed as a more tractable alternative to inlining with less drastic resource costs. However, it is difficult to model many important analyses in an interprocedural setting, and many of the analyses degrade markedly in the usual case where not all program source is visible to the analyzer. Even if interprocedural analysis is performed, effective use of this information will almost always require code expansion, since many of the code transformations enabled by an interprocedural analysis are impossible to safely express without some duplication of code in either the caller, the callee, or both.

Our high-level intermediate-code optimizer, HLO, employs both inlining and cloning in combination to achieve its optimization goals. Cloning is goal-directed: it is used to expose particularly important details about the calling context to the callee. Inlining is used more liberally to allow traditional optimizations to affect a wider scope. HLO's inlining and cloning capabilities are uniquely powerful: it can inline or clone calls both within and across program modules, can inline or clone independent of source language, can accommodate both user directives and profile directed feedback, and can inline or clone at almost every call site with very few restrictions.

The aggressive inlining and cloning done by HLO can substantially reduce the run time of a program. For example, on the six SPEC92 integer benchmarks, HLO's inlining and cloning boost the overall performance ratio by a factor of 1.24 on a PA8000 workstation, with a maximum speedup ratio of 2.02. For the eight SPEC95 integer benchmarks the results are even more dramatic, boosting overall performance by a factor of 1.32 with a maximum speedup ratio of 1.80.

The remainder of this paper is organized as follows: Section 2 describes the capabilities and structure of HLO; Section 3 presents data from a number of measurements; Section 4 describes related work, and Section 5 summarizes our results and describes opportunities for future work.

2 HLO's Inlining and Cloning

2.1 Compiler Infrastructure

The current generation of HP compilers communicate via a common intermediate language known as *ucode*. Language front-ends produce ucode, and the common back end accepts ucode as input. HLO acts as a ucode-to-ucode transformer interposed between the front and back ends of the compiler. By buffering the ucode from the front-end, HLO is able to perform module-at-a-time optimizations. An alternative compile path allows the ucode to be stored into special object files known as *isoms*. These files remain unoptimized until link time. When the linker is invoked and discovers isoms, it passes them en masse to HLO, which performs optimizations and then passes the files one-at-a-time to the back end, where real object files are produced. After all files have been optimized the linker is reinvoked on the real object files to build the final executable. This path allows HLO to perform intermodule optimizations.

The isom path is fully `make` compatible. It also allows for the incorporation of profile information — for example, branch execution counts — gathered by previous training runs. The availability of profile information feeds the inlining and cloning heuristics, and enables a number of other profile-based optimizations (PBO) within the compiler [15, 9, 12]. Figure 1 gives a picture of how all this fits together.

2.2 Structure of HLO

Conceptually, HLO operates as something of a pipeline. The input stage translates the ucode into HLO's own internal representation (IR), and builds up a comprehensive symbol table. A variety of classic optimizations (e.g. constant propagation) are performed on the IR at this time, mainly to reduce its size. After all code has been input, a limited amount of interprocedural analysis is performed. HLO then inlines and clones in a manner we describe below. The output phase converts the HLO IR back into ucode and sends the ucode on to the back end for intensive intraprocedural optimization and ultimately generation of object code.

HLO performs several passes of inlining and cloning. The main motivation for this multi-pass structure is that it is quite difficult to anticipate the optimization impact of a particular inline or clone. If all inlining and cloning were done in a single pass HLO would not be able to focus in on particular areas of interest revealed only after the first stage of inlining. Having multiple passes also simplifies matters like cloning a recursive procedure with a pass-through parameter which might be difficult to do correctly in a single pass. The overall algorithm is sketched in Figure 2.

High-level control of the inliner is done by giving the inliner a budget. This budget is an estimate of how much compile time will increase because of inlining. By default the inliner will try to limit compile-time increases to 100% over no inlining. Note that because various optimization phases are nonlinear, a 100% increase in compile time does not imply that the inliner will double the size of the code. The HP-UX backend optimizer contains several algorithms that are quadratic in the size of the routine being optimized, so we model this effect accordingly. For our compiler, then, code growth is typically on the order of 20%. The budget can be adjusted in either direction by a variety of user controls. Once the overall budget has been computed, the inliner computes the *staging* for the budget. This apportions the budget amongst the various passes, basically to ensure that not all of the budget is used up in the first pass. The compiler then alternates cloning and inlining passes until either the budget is exhausted or a pass limit is reached.

2.3 Cloning

Cloning begins with the selection of cloning sites. Each call site is examined in turn. The cloner first determines if the call site passes certain legality tests. For example, cloning is disallowed if there are gross type mismatches between caller and callee, or if the caller and callee do not agree on the number of parameters to be passed.[1]

Next, the cloner determines if the caller supplies interesting information to the callee. For example, the caller might pass an integer 0 as the first actual parameter. If the calling context is sufficiently interesting, the callee's use of this context is queried next. If the callee can benefit from knowing about its formals what the caller knows about its actuals, the site is considered to be one suitable for cloning. At this point, the

[1]We could clone even in such cases, but the idea is to try and preserve the behavior of even semantically incorrect programs.

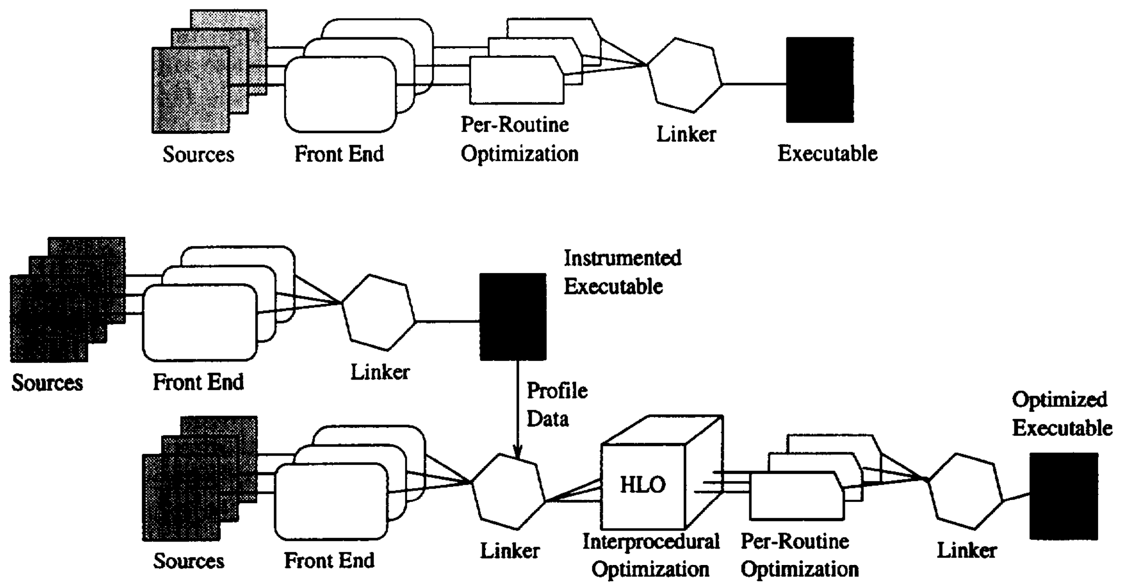

Figure 1: (Top) A traditional compile path, supporting intraprocedural optimization, and interprocedural optimization within a source module. (Bottom) The path used in HP-UX compilers to support cross-module optimization and profile-based optimization.

```
Inline and Clone(G)

INPUT
  call graph G: (routines, edges)

ALGORITHM
  // estimate current compile time cost
  current cost C = 0
  FOREACH routine R IN G
    C = C + $(\text{sizeof}(R))^2$

  // determine budget
  growth factor D = 1.2
  budget B = C * D

  // determine staging of budget
  S[0] = C + B * 0.2
   ...
  S[limit-1] = C + B

  // inline and clone
  clone database D = { }
  pass number P = 0
  WHILE ( C < B AND P < limit ) DO
    C = Clone(G,S[P],C,D)
    C = Inline(G,S[P],C)
    P = P + 1
```

Figure 2: Overall Inlining and Cloning algorithm

cloner effectively intersects the information supplied by the caller and the information useful to the callee to create a clone specification, or *clone spec*. In our current implementation, only caller-supplied constants are considered interesting. Many other criteria are possible: cloning to exploit aliasing properties that hold at the call site, or the fact that certain arguments are ignored by the callee, or that the caller ignores the return value, and so on.

Our current implementation of the callee-side analysis is relatively simplistic. Each parameter is considered independently, only the abstract constancy or non-constancy of the parameter is considered, and we do not model interprocedural effects (pass-through constants). Special emphasis is put on parameter values that reach the function position at an indirect call site. Each interesting use of a parameter is weighed by an estimate of the importance of that use. When PBO data is present, the compiler computes the profile count of the block relative to the routine entry; without such data it uses heuristics to guess at the relative importance.

After finding an interesting call site, the cloner could continue on building clone specs for all suitable call sites, but doing so might lead to unnecessary proliferation of clones. Instead, once an interesting site has been found, the cloner uses the clone spec to try and greedily create a *clone group*: a set of call sites which can safely call the clone described by the clone spec. This is done by examining each of the calls made to the callee to see if the calling context at the call site is compatible with the clone spec created for the clone group. If so, the call site is included into the clone group. Once the clone group is completely formed, the cloner then assesses the run-time benefit of making the clone. This calculation takes into account factors like the estimated total number of calls that will call the clone instead of the original routine, and the value to the callee of the caller-specific context information.

After all call sites have been examined, the cloner has a collection of clone groups describing the particular clones that could be created, and an estimate of the benefit of creating each clone. The cloner then ranks all clone groups by benefit and greedily creates clones and modifies call sites until the current allotment of the compile-time growth budget has been used up. Any clone groups that were not handled in this pass are discarded; they may be recreated and cloned in a later pass.

Creation of a clone is fairly straightforward. The IR for the clonee is duplicated, and any formal parameters that are known from the calling context are turned into routine-scope variables and initialized with appropriate constants in the clone's entry block. If there are slight discrepancies in type, a type cast is inserted. The clone is always placed into the same module as the clonee. If the caller is in another module and is passing symbolic information which is only visible in the caller's module (e.g. the address of a file-static procedure), this information must be promoted to global scope and given a unique name that will not collide with any user-supplied name.

As clones are created, the clone and associated clone spec are also recorded in a special database. This database comes into play in later cloning passes, when it is possible that the cloner will reproduce the same clone spec used to clone in an earlier pass, because intervening optimizations have sharpened the information available at call sites which were previously not worth consideration. If a given clone exists in the database then it is simply reused; otherwise the clone must be created as described above.

Modification of the call sites in a clone group to invoke the clone is also fairly straightforward. The clone spec describes the signature of the new routine, so any parameters incorporated into the clone are edited from the actuals list. The call site is then modified to refer to the clone instead of the original routine. This modification in turn inspires changes in the call graph to reflect the new relationships between caller, clonee, and clone. In particular, if all calls to a clonee are replaced by calls to a clone, the clonee may become unreachable in the call graph and will be deleted. The cloner attempts to anticipate subsequent clonee deletion when estimating

```
Clone(G,B,C,D) : returns C

INPUT
 call graph G: (routines,edges)
 budget B
 current cost C
 clone database D

ALGORITHM
 // setup
 FOREACH routine R IN G
  create parameter-usage descriptor P(R)
 FOREACH edge E IN G
  create calling-context descriptor S(E)

 // build clone groups
 FOREACH edge E in G
  callee R = E.target
  IF ( clonable(R) and clonable(E) ) THEN
   clone spec CS = intersect( S(E), P(R) )
   IF ( CS is nonempty ) THEN
    clone group CG = (R,CS,E)
    FOREACH edge E' incident on R
     IF ( clonable(E') AND
          matches( S(E), CS ) ) THEN
      add E' to CG
    estimate benefit of CG

 // select clones
 sort CGs by benefit; C' = C
 FOREACH clone group CG IN CGs
  cost X = $(\text{sizeof}(R))^2$
  IF ( C' + X < B ) THEN
   accept CG; C' = C' + X

 // create clones and fix call sites
 FOREACH clone group CG IN accepted CGs
  IF ( ! lookup(D, R, CS) ) THEN
   R' = make clone (R, CS)
   add database entry ( R, CS, R' )
  FOREACH edge E' IN CG
   change target of E' from R to R'

 // optimize clones and recalibrate
 FOREACH newly created clone R'
  optimize(R')
  C = C + $(\text{sizeof}(R'))^2$
```

Figure 3: Cloning Pass

the budget impact of a particular clone group or groups; in effect, a clone group that ensures that the clonee will be deleted is considered to have no compile time impact.

2.4 Inlining

The overall structure of an inlining pass is similar to cloning. The inliner first considers all call sites for any legal, technical, pragmatic, or user-imposed restrictions on inlining. Illegal sites include those with gross type mismatches, varargs, or argument arity differences. Technically restricted sites include those where information specific to the callee disagrees with information specific to the caller. For example, the caller's IR may specify that re-association of floating point operations is allowed, while the callee's IR may indicate that such re-associations are unacceptable. By and large these kinds of restrictions are imposed to simplify the task of representation of this information. Pragmatic concerns include issues like handling callees that use `alloca` to dynamically allocate space on the stack, or inlining at a site where actual parameters describe overlapping regions of memory and the callee is allowed to assume that its formal parameters do not alias. User imposed restrictions come from various command line options and pragmas.

Once the set of viable inlining sites has been identified, they are assigned a runtime figure of merit. High-frequency call sites are given highest priority. Sites that occur in blocks executed less frequently than the routine entry block are assigned a penalty. This helps to avoid inlining into a non-critical path; doing so might cause increases in register pressure which push spills into critical code paths and hurt performance.

The inliner then walks over the inline site list in priority order. The compile-time impact of each site is considered, and if within the current budget, the inline is accepted. Computation of the compile time effect is complicated by interactions among inlines. For example, if `A` calls `B` and `B` calls `C`, the cost of inlining `B` into `A` depends on whether or not `C` has been already been inlined into `B`. To model this dependence, the inliner keeps a schedule of the order in which it will perform all accepted inlines. By and large, the inliner attempts to work bottom-up over the call graph. To compute the cost of inlining `B` into `A`, a description of the inline is first inserted into the schedule in the appropriate spot. If `B` is then determined to be the target of an earlier inline or inlines, the estimated size of `B` after those inlines have been performed is used to compute the cost of optimizing `A`.

The inliner processes and accepts call sites greedily until its allotment of the budget is exhausted. At this point the remaining viable inline sites are discarded

```
Inline(G,B,C) : returns C

INPUT
 call graph G: (routines,edges)
 budget B
 current cost C

ALGORITHM
 // screen inline candidates
 FOREACH edge E IN G
  IF ( inlineable(E) ) THEN
   accept E; compute benefit(E)

 // select inline sites
 sort accepted E's by benefit
 C' = C
 FOREACH accepted edge E
  insert E into schedule
  cost X =
   (sizeof( E.target + E.source ))^2
     - (sizeof( E.target ))^2
  C'' = C'
  C' = C' + X
  IF ( E.target is source in
       later inline ) THEN
    adjust C' for cascaded cost
  IF ( C' > B ) THEN
    remove E from schedule
    C' = C''

 // perform inlines
 FOREACH scheduled edge E
   inline E.target into E.source

 // optimize inlines and recalibrate
 FOREACH routine inlined into R'
  optimize(R')
  C = C + (sizeof(R'))^2
```

Figure 4: Inlining Pass

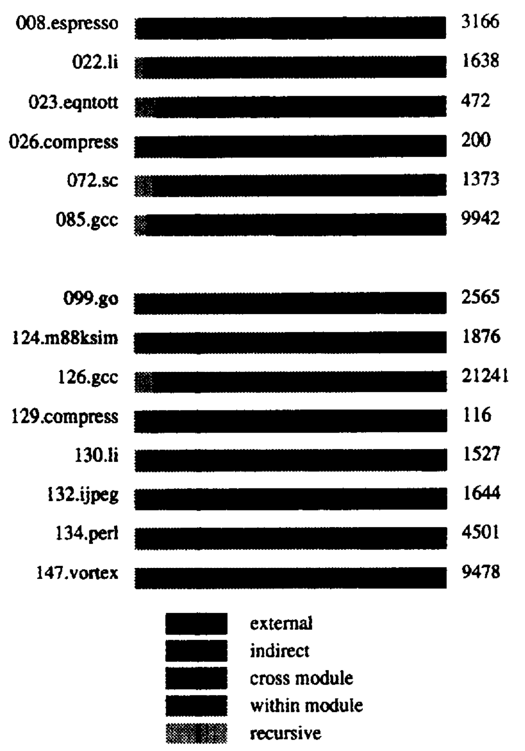

Figure 5: Static characteristics of call sites in the SPEC integer benchmarks. The number at right is the total number of call sites in the code.

(they may be reconsidered in a subsequent pass of inlining). The inliner then uses its schedule to carry out each inline in the list of accepted inlines. As with cloning, movement of code between modules may result in program entities being promoted to wider scopes.

3 Measurements

3.1 Characteristics of Call Sites

Figure 5 illustrates some static information about the 14 programs in the SPEC92 and SPEC95 suites. Each call site in these programs can be classified into one of five categories: external, indirect, cross-module, within-module cross-routine, and recursive.

External sites represent calls to library routines or to program modules not visible to the compiler. In principle it is possible to provide intermediate code versions of such libraries and modules to broaden the scope of inlining or cloning even further, but the results reported in this paper are with standard precompiled libraries, with one notable exception. The `072.sc` benchmark includes a special `curses` library in which all curses calls do nothing. These calls (reported in our figure as cross-module calls) would be ideal candidates for inlining, but they are eliminated before inlining because HLO's interprocedural analysis determines that they have no side effect.

At indirect sites the callee is computed at run time, so these sites are not directly amenable to inlining or cloning. It is possible to employ various techniques to try and resolve the target of indirect calls at compile time. For example, HLO will aggressively clone at sites where the caller passes a pointer to a procedure and the callee uses the value of a formal variable in an indirect call. Subsequent constant propagation of this code pointer to the call site will then provide the information needed to turn the indirect call into a direct call, which can then be inlined or cloned in a later pass. This sort of staged optimization would be much more difficult to accomplish in a single inlining pass.

The remaining are amenable to inlining and cloning. As the figure shows, there are significant numbers of cross-module calls. The ability to inline these cross-module calls is crucial for good performance.

3.2 Transformations to SPEC Integer Programs

Table 1 shows more detail on the transformations done to a subset of the SPECint programs by HLO. There are several points worthy of further discussion. First, as more information is made available to the compiler, the quality of the code improves. For instance, in `072.sc`, the base performance level with inlining and cloning done per-module is 7.1 seconds. If the compiler is allowed to inline and clone cross-module, the runtime drops to 6.3 seconds. If the compiler is allowed to make use of profile information, the runtime becomes 5.3 seconds. Finally, the combination of both cross-module inlining and cloning with profile feedback gives a run time of 4.5 seconds. By and large, this monotonic improvement property holds for almost all programs that we have examined.

Another consequence of the increase in scope is that compile time[2] increases. Again looking at `072.sc`, the base compile time is 862 seconds, while the compile time with cross module inlining and profile feedback is 1786 seconds, approximately 100% larger (this time includes the time required for the instrumenting compile, training run, and final compile). In some cases, the compile time increases are a good deal larger; in others, smaller. The precise impact is often difficult to estimate because the analyses performed downstream are often quite sensitive to particular sorts of code structures.

[2]All programs were compiled on an HP K400 workstation using special developmental versions of the HP-UX 10.20 compilers. The times shown are therefore 30-40% slower than would be obtained by production compilers on the same hardware.

Benchmark	Scope	Inlines	Clones	Clone Repls	Deletions	Compile Time	Run Time
008.espresso		281	28	47	28	826	8.6
	c	188	18	32	23	983	8.2
	p	815	45	106	9	976	8.2
	cp	297	17	47	7	1047	7.6
022.li		256	42	52	63	348	25.6
	c	76	13	18	15	466	20.0
	p	620	93	495	35	323	23.0
	cp	90	23	256	10	973	11.4
072.sc		127	26	30	18	862	7.1
	c	39	6	8	4	981	6.3
	p	244	42	50	21	1850	5.3
	cp	106	12	17	6	1786	4.5
085.gcc		732	87	247	70	3814	22.6
	c	1008	230	760	193	13544	22.5
	p	309	47	110	25	1210	22.6
	cp	641	349	2484	80	17513	22.0
099.go		400	23	219	6	693	465.1
	c	545	30	371	2	877	453.4
	p	154	14	177	0	1013	436.3
	cp	121	22	327	0	996	386.0
124.m88ksim		140	33	64	18	491	298.0
	c	339	121	431	19	702	284.8
	p	97	21	27	17	783	228.7
	cp	80	49	132	7	971	177.4
147.vortex		253	17	67	9	2228	446.7
	c	841	121	2211	5	2320	445.3
	p	140	9	12	5	2028	373.7
	cp	175	83	2142	1	2522	270.1

Table 1: Inline and clone information for selected benchmarks. Here c indicates cross-module compilation, p profile-based compilation. Baseline is a compiler with full inlining and cloning capabilities.

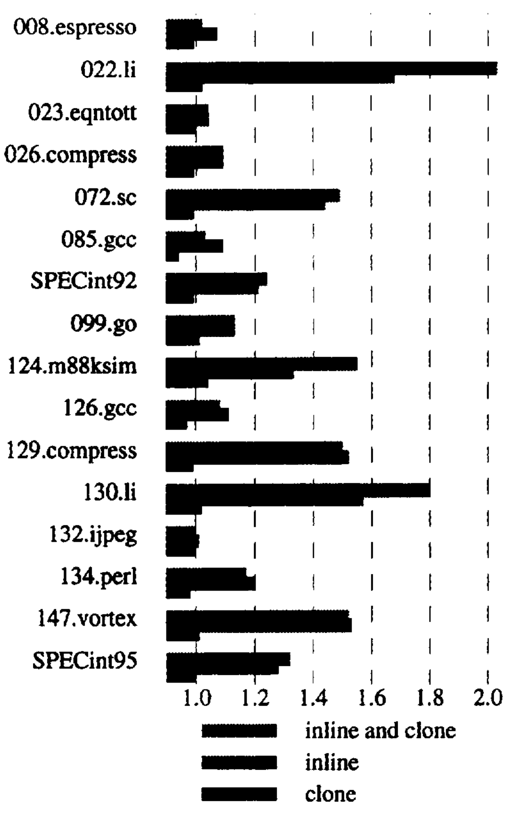

Figure 6: Relative speedup of SPEC integer programs with inlining, cloning, or both, for the PA8000 workstation. Baseline compile uses cross-module and profile-based optimization, plus peak options not affecting inlining or cloning. Overall figures present the geometric mean speedup for each benchmark suite.

Data in this table also underscores the role of clone groups. Most clones are usable at more than one call site, and as the optimization scope widens, the ratio of call sites modified to clones created increases, indicating that a given clone is being used at a larger number of sites on average. The distribution tends to be quite skewed. A sizeable number of routines are also deleted during compilation. These include both file-scope user routines and clones which are provably not callable because all calls have either been cloned or inlined.

The data in table also indicates the synergistic benefits of profile-based and cross-module optimizations. For instance, in `099.go`, the cross-module profile-based compilation actually does fewer inlines than the other compilations, yet produces a faster binary. Compile times are shorter than the cross-module alone case, despite the need for a preliminary compile with instrumentation and a training run to produce the profile database. The data also shows that for our compiler, profile-based optimization is usually more valuable than cross-module optimization. We cannot yet say if this represents anything fundamental; it may simply be an indication of the relative maturity of the profile-based optimization components.

3.3 Overall Performance

Figure 6 shows the relative performance of the 14 SPECint programs as measured on a PA8000-based [10] K460 workstation running HP-UX 10.20. The workstation had two 180 MHz cpus and 256 MB of memory, 16-way interleaved. Programs were compiled with the HP-UX 10.20 C compiler. All compilations used interprocedural optimization (`+O4 +Onolimit`) and all the compiles incorporated profile information (`+P`) gathered from an instrumentation run done on the specified training data set. Each benchmark was compiled four separate times: with no inlining or cloning, with only inlining or only cloning, and with both inlining and cloning. Each executable was run three times on an unloaded workstation, and the best time reported was used.

The data shows that inlining alone has the biggest impact on performance, though cloning is a vital contributor to both `022.li` and `130.li` (which are quite similar) and to `124.m88ksim`. Cloning by itself does not yield significant performance improvements, and on some benchmarks actually reduces performance slightly over what can be obtained by inlining alone. Though we have several theories, we have not as yet been able to determine the precise reason or reasons for the performance losses seen in some benchmarks when just cloning is used.

What is it that happens in inlining that leads to these speedups? To try and answer this question we ran several of the benchmarks through a PA8000 simulator. Data for several of these sets of simulations are presented in Figure 7. In gathering this data, the simulator ran modified versions of the SPEC95 integer benchmarks, with simplified input sets designed to closely mimic the behavior of the benchmark.

The simulation data shows that in several benchmarks inlining has resulted in dramatic drops in overall execution time (as measured by cycles) and the number of instructions retired by the processor. The effect on the CPI varies; in `130.li` it falls dramatically, but in `147.vortex` it rises; yet both benchmarks speed up substantially.

Not surprisingly, inlining and cloning both tend to increase the I cache miss rate and the total number of I cache misses. For the most part, however, inlining reduces the total number of I cache accesses, meaning

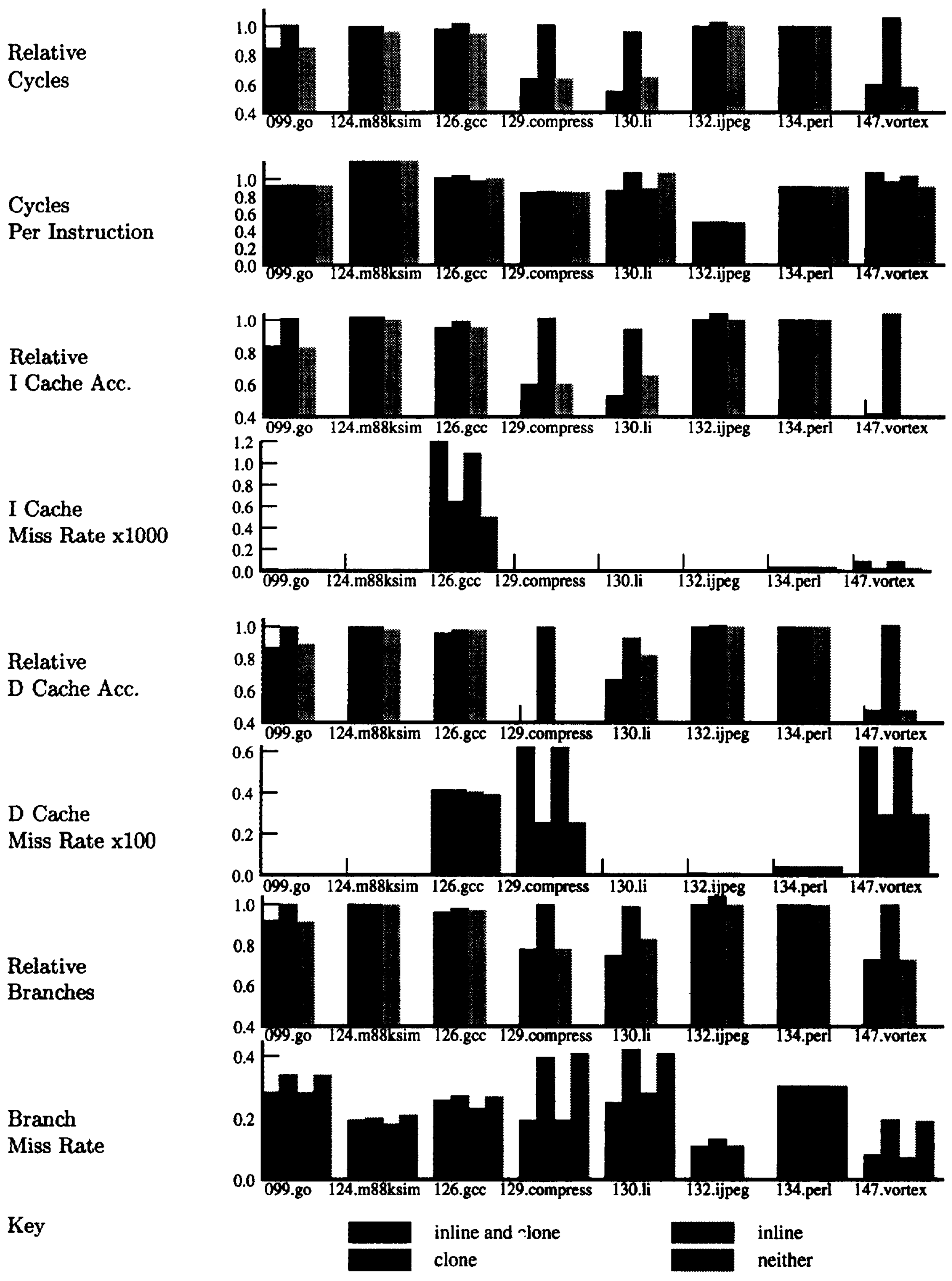

Figure 7: Simulation results for the PA8000 running a modified versions of the SPEC integer benchmarks. *Relative* indicates that the data is scaled relative to the run with neither inlining or cloning.

that some of the increase in miss rate is due to the same number of I cache misses being amortized over fewer accesses. In the larger benchmarks, especially 126.gcc, the I cache miss rate more than doubles. The overall impact of the inlining and cloning on I cache performance is unclear, and seems to depend on the particular dynamic of the program in question.

The number of D cache accesses is also dramatically decreased. This causes an increase in the data cache miss rate, again because a similar number of misses is spread over fewer total accesses. A big part of this dramatic drop is the elimination of caller and callee register save operations at call sites that have been inlined. We believe that this indicates that the register allocation phase of the HP-UX compiler has little difficulty with the larger routines created by inlining and cloning, and that for the most part register pressure is not an issue.

The number of branches overall is reduced. Since this number includes procedure calls this is also not surprising. The branches that remain appear to become more predictable. Since the PA8000 always mispredicts procedure return branches, this may also be a misleading statistic. However, we suspect that the predictability of the remaining branches is also enhanced. Any possible improvement in branch behavior must be weighed off against an increase in the total number of branches, which may increase the rate of branch collision in a branch prediction cache.

3.4 Validation of Heuristics

There is no practical technique at compile time for determining the optimum set of inlines or clones. Both the run time benefit and compile time costs must be estimated (see Ball [3] for an example). Furthermore, in any reasonably sized benchmark, there are a staggering number of ways to perform inlining and cloning. Chang, Mahlke, Chen, and Hwu [5] point out that a simplified version of the problem is equivalent to the knapsack problem, which is known to be NP complete. Each site can be either inlined or cloned, and the order of inlining and cloning may make a difference in the final code.

Inlining and cloning must thus be guided by heuristics. As with all heuristics it is important to verify that the decisions they make are reasonably sound ones. To this end we present the data shown in Figure 8, which illustrates one technique for assessing the quality of heuristics used in HLO. As an experiment, we varied the budget provided to the inliner from a relatively small budget of 25 to a large budget of 1000. For each budget level we compiled the benchmark 022.li a number of times. In each compile we artificially stopped the inliner after a certain number of inlines and/or clone replacements. The resulting curves depict the incremental benefit of each successive inline or clone replacement. As can be seen, very few inlines or clones have an adverse impact on performance. Also, once the budget has reached a sufficiently large value (100 in this case), there is no additional performance increase with extra inlining. This property (that performance reaches an asymptote with increasing budget) is true of many of the programs we have studied. Our default budget of 100 was chosen to maximize the performance of the benchmarks we studied without performing unnecessary inlines.

3.5 Inlining in Other Codes

Inlining is also of potential benefit in any language that supports the notion of a procedure or subroutine. At present, HLO is only capable of optimizing C, FORTRAN, and C++ programs.

In this paper, we do not report any data for the floating-point benchmarks in the SPEC suite. One reason for this is that there is a significant barrier to inlining in FORTRAN: it is difficult to aggressively represent the aliasing semantics of an inlined FORTRAN subroutine. By default, the compiler is free to assume that formal parameters do not alias each other nor any global variables. Representing this information in the post-inlined code is tricky, and compilers (like ours) that cannot properly represent it are usually better off not inlining.

The SPEC benchmarks tend to be small programs; 126.gcc is the largest at around 120,000 lines of code. A major challenge to effectively deploying aggressive inlining is the sheer size of production codes. We have recently been experimenting with compiling the 500,000 line performance kernel of an important application program, and have been amazed to find that significant speedups like we see in some of the SPEC benchmarks can also be obtained in large production codes.

4 Related Work

Many research and production systems have been capable of inlining and cloning [1, 2, 14, 7, 5, 4, 11, 6, 8, 13]. However, very few have reported consistently good results in a mature compilation system, reported results on moderately large well-known programs, reported the effects of aggressive inlining and cloning, or offered a detailed analyses of the costs and benefits of inlining.

Chang, Mahlke, Chen, and Hwu [5, 11] describe a profile-based inliner for the IMPACT compiler. Their work is perhaps closest in spirit to ours, inlining across modules, and making use of profile information to select the best inlining sites. The control algorithm makes a

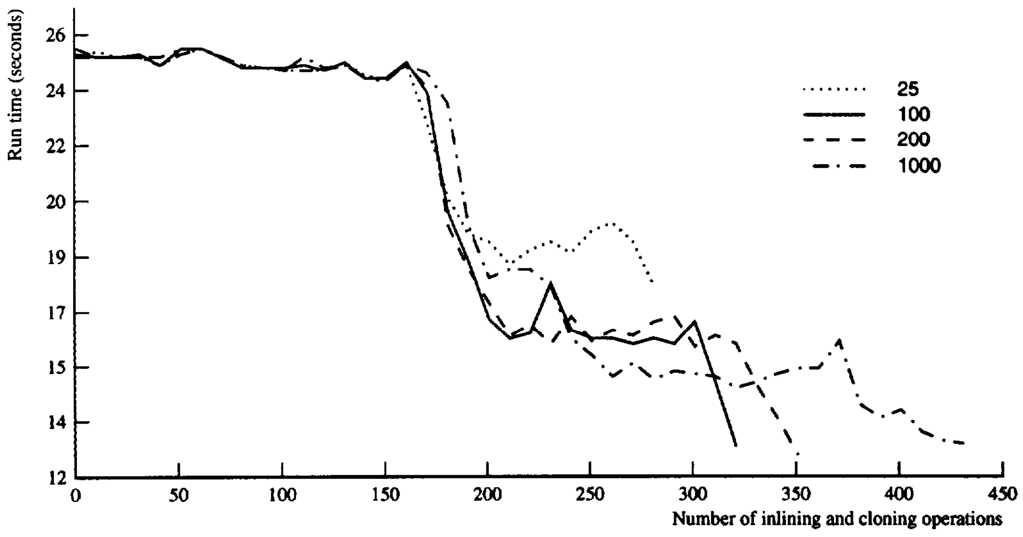

Figure 8: Incremental benefit of inlines and clone replacements in 022.li, at various budget levels.

single pass over the inlineable sites, ranking them by profile weight. As in our implementation, selected inlines are then scheduled to be performed in roughly bottom-up call graph order. Overall control is governed by a code growth budget. They report a mean speedup of 11% with a maximum speedup of 46%. Our work differs in a few important aspects: we make use of cloning, perform multiple passes, rank inline sites both in terms of profile weight and relative execution frequency, and have profile information not only on interprocedural arcs but also intraprocedural ones.

Davidson and Holler [7] developed an inliner for C programs that operated at source level. They reported a mean speedup of about 12% and a maximum speedup of about 35% on a variety of programs. They noticed a number of cases where inlining induced register pressure limited performance. Their study was done without the benefit of profile information, which we believe to be crucial to getting good performance.

Allen and Johnson [2] describe a C language inliner and give a good discussion of some of the motivations for inlining. However, we feel that the commonly held notion (found in [2] and elsewhere) that an inliner should aim to inline only small functions to be untrue. Medium and large functions should be inlined if the remainder of the compile path is capable of aggressively handling large functions.

Cooper, Hall, and Kennedy [6] describe a cloning algorithm which is a good deal more sophisticated than ours. Their analysis is interprocedural and relies on the actual values of constants passed to callees. Our use of clone specifications and clone groups mimics some of the clone vector merging possible in their work. For reasons we do not yet completely understand, we have found our implementation of cloning to be relatively ineffective in boosting performance.

Dean and Chambers [8] describe an interesting system that is able to perform experiments to determine the actual benefits of an inline. Our system is handicapped by having to rely on static estimates of the benefit of an inline, and assessing an independent benefit of any particular inline is not easy, since such benefits depend upon all the other inlining decisions that have been made so far. For code under development in situations where development builds make use of inlining, however, the idea of a database to record information about past inlining decisions is appealing.

5 Summary and Future Work

Our experiences with inlining and cloning in HLO demonstrate that aggressive inlining and cloning can give substantial and widespread improvements in the performance of programs, with some well-studied programs like 022.li speeding up by a factor of two.

Our inliner differs from previous inliners described in the literature in that it is able to inline at almost any call site without restriction; it can inline cross-module and cross-language calls; it is uses profile feedback in conjunction with multiple passes to aggressively inline in the important parts of the program and adapt to the consequences of previous inlines and clones; and it keeps a budget that allows for a global assessment of compile time impact without artificially restricting the amount of inlining in any one portion of the code.

Our inliner was added to a mature compiler that already contained an aggressive, state-of-the-art global optimizer. The fact that inlining produces such impressive additional speedups is an indication that the gains we are seeing here are not simply straw-man artifacts where high-level optimizations eliminate problematic code from an immature global optimizer. Instead, the inliner's actions expose more significant and weighty regions of code to the global optimizer, and thereby enable the full power of the compiler to be trained on the performance-critical portions of the application.

Though we are pleasantly surprised with the results we have obtained so far, we have a number of future projects in mind. We want to apply aggressive inlining to large, production programs like the HP-UX kernel, database applications, and CAD tools. We also plan to improve the impact of cloning and remove the inlining restrictions for FORTRAN codes. We are looking at techniques to make profiling less onerous, perhaps incorporating profile information from a variety of sources. We are also contemplating using aggressive outlining as a complement to aggressive inlining, to help further focus the global optimizer on the truly important stretches of code.

Acknowledgments

We gratefully acknowledge the inspired contributions made by our colleagues on the HLO project and in other HP language projects, especially Anne Holler, Manuel Benitez, Wei Hsu, Lacky Shah, Carl Burch, Rajiv Kumar, Adam Matusiak, John Liu, Jonathan Springer, and Steve Rehrauer. We also appreciate the thoughtful comments of the anonymous referees which helped to substantially improve this paper.

References

[1] F. C. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, R. Ruskin, Ed., Prentice-Hall, Englewood Cliffs, NJ, 1971, 1-30.

[2] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 241-249.

[3] J. Ball. Predicting the effects of optimization on a procedure body. *ACM SIGPLAN Notices 14(8)*, 214–220, 1979.

[4] P. Carini. *Automatic Inlining.* IBM Research Report RC 20286, November 1995.

[5] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience 22(5)*, 349-369, May 1992.

[6] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages, 19(2)*, 105-117, February 1993.

[7] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software Practice and Experience 18(8)*, 775-790, August 1988.

[8] J. Dean and C. Chambers. Training compilers to make better inlining decisions. Technical Report 93-05-05, Department of Computer Science and Engineering, University of Washington, 1993.

[9] A. M. Holler. Compiler optimizations for the PA-8000. *COMPCON 1997 Digest of Papers*, February 1997.

[10] D. Hunt. Advanced performance features of the 64-bit PA8000. *COMPCON 1995 Digest of Papers*, 123-128, March 1995.

[11] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 246-257.

[12] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 16-27.

[13] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters, 32(3)*, 137–142, August 1989.

[14] R. W. Schiefler. An analysis of inline substitution for a structured programming language. *Communications of the ACM 20(9)*, 647-654, September 1977.

[15] S. E. Speer, R. Kumar, and C. Partridge. Improving UNIX Kernel Performance using Profile Based Optimization. In *USENIX 1994 Proceedings.*