Journal of Instruction-Level Parallelism 3 (2000) 1--25 Submitted 2/2000; published 4/2000

Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha

Robert Cohn ROBERT.COHN@COMPAQ.COM
P. Geoffrey Lowney GEOFF.LOWNEY@COMPAQ.COM
Compaq Computer Corporation
334 South Street
Shrewsbury, MA 01749

Abstract

This paper describes and evaluates the profile-based optimizations in the Compaq C compiler tool chain for Alpha. The optimizations include superblock formation, inlining, commando loop optimization, register allocation, code layout, and switch statement optimization. The optimizations either are extensions of classical optimizations or are restructuring transformations that enable classical optimizations. Profile-based optimization is highly effective, achieving a 17% speedup over aggressive classical optimization on the SPECInt95 benchmarks. Inlining contributes the most performance and code layout, superblock formation, and loop restructuring are also important.

1. Introduction

When tuning programs, we often notice that the compiler has made poor optimization decisions. Compilers can only use the information they are given, and we usually know much more about a program than is expressed in the source code. One important piece of information is the execution behavior of a program. How many times does the loop iterate? Is a load likely to miss in the first level cache? Is an *else* clause likely to be executed?

Profile-based optimization (PBO) is a way to give the compiler information about the run-time behavior of a program. The program is profiled, and this information is used by the compiler

to generate code, often making frequently executed paths through the code faster and other paths slower.

This paper describes and evaluates the profile-based optimizations that are used in the Compaq C compiler tool chain for Alpha. PBO was added to a mature compiler with a very powerful and complete classical optimizer. We tried to leverage this optimizer where possible. Many of the classical optimizations were already driven by a cost model; some of the cost models included estimates of execution counts. It was usually a straightforward task to extend the cost model to use execution counts or to replace the estimated counts with measured ones. In some situations, the desired code improvement can not be achieved directly by classical optimization. In these cases, we use profile information to drive a restructuring transformation of the code, making it possible for the classical optimization to do the rest of the work. The result is a compiler that achieves large speedups with PBO, but only a small percentage of the code is specific to PBO.

This paper is unique in that it evaluates profile-based optimizations in the context of a product compiler with a high quality classical optimizer. We measure how individual optimizations contribute to the overall speedup. While most of the optimizations are based upon previous work, we describe how to adapt them to fit more naturally into an optimizer, how to generalize them so that they provide more consistent speedups, and their payoff in increased performance.

In Section 2, we describe the compilation tools. We present the details of the profile-based optimizations in Section 3. The optimizations are evaluated in Section 4. We summarize our findings in Section 5.

2. Background

Profile-based optimization is performed in the GEM compiler back end [8] and in Spike [9], a post-link optimizer. The GEM back end is used by the C, C++, and FORTRAN compilers on Compaq Tru64 Unix, OpenVMS, and Microsoft Windows NT. It generates highly optimized

code for Alpha CPUs that significantly outperforms code generated by gcc [1]. Most of the profile-based optimization in GEM is performed in the optimizer, which transforms the IL (intermediate language representation) generated by the front end into a semantically equivalent form that executes faster on the target machine. The rest is done in the code generator, which translates IL into machine instructions and allocates registers.

Spike optimizes Alpha executables and shared libraries for Tru64 Unix and Windows/NT. It uses profile information for code layout and instruction alignment.

The ideas and algorithms for our profile-based optimizations were taken from many different sources, and we discuss their relationship to prior work in Section 3.

3. Optimizations

Profiles are collected by either instrumenting a binary with pixie [2] or using the statistical sampling profiler DCPI [14]. Both produce a database of basic block execution counts. The compiler reads the profile database and annotates basic blocks in its IL with execution counts. It computes call edges counts and estimates flow edge counts from this information. This is the only profile information that is used. Optimizations that alter the flow graph must update the execution counts as appropriate. The system that manages profiles is described in detail by Albert [23].

3.1. Inliner

Procedure calls can be barriers to optimization. Inlining eliminates that barrier as well as the invocation overhead by replacing the call with a copy of the body of the function. It can also improve spatial locality, which is especially useful when the processor does next line instruction prefetching.

When profile information is not available, the compiler performs inlining using an algorithm that estimates the positive and negative effects of inlining a function. The negative effects that are considered are code growth, loss of temporal locality in the instruction cache, and register pres-

sure. The positive effects considered are better optimization from the elimination of calls and replacement of arguments with constant values. Profile information is used to adjust the desirability of inlining, but the algorithm is otherwise unchanged. Functions that are rarely executed are much less likely to be inlined, and the compiler is more willing to inline large functions if they are frequently executed. A function is more likely to be inlined by the static inliner if it is only called by one procedure, as this increases spatial locality in the instruction cache without a decrease in temporal locatlity. When profile information is present, this heuristic is extended to functions where almost all of the invocations are from the same caller.

3.2. Tracer

The tracer converts paths through complicated control flow, including short trip count loops, into large superblocks through a combination of superblock formation and loop peeling [15]. Larger superblocks improve scheduling in the GEM compiler, which can only schedule a single superblock at a time. Superblock formation also restructures the code so that the compiler can ignore the effects of infrequently executed paths.

Loop peeling is a transformation that pulls one or more copies of the loop body outside of the loop. Loop entrances and exits are barriers for some optimizations. For loops that only execute a small number of iterations, it is advantageous to peel off a few iterations because it allows the compiler to optimize code before and after the loop with the loop body. If a loop executes more than a few iterations, unrolling is better than peeling.

The tracer uses flow edge counts to select a trace, which is a frequently executed path through the flow graph [11]. Trace selection [11], [12] starts with a seed flow-graph edge. The trace is then grown forwards and backwards using the mutual-most likely heuristic. The heuristic requires that for block A to be followed by block B in the trace, A must be B’s most likely predecessor and B must be A’s most likely successor.

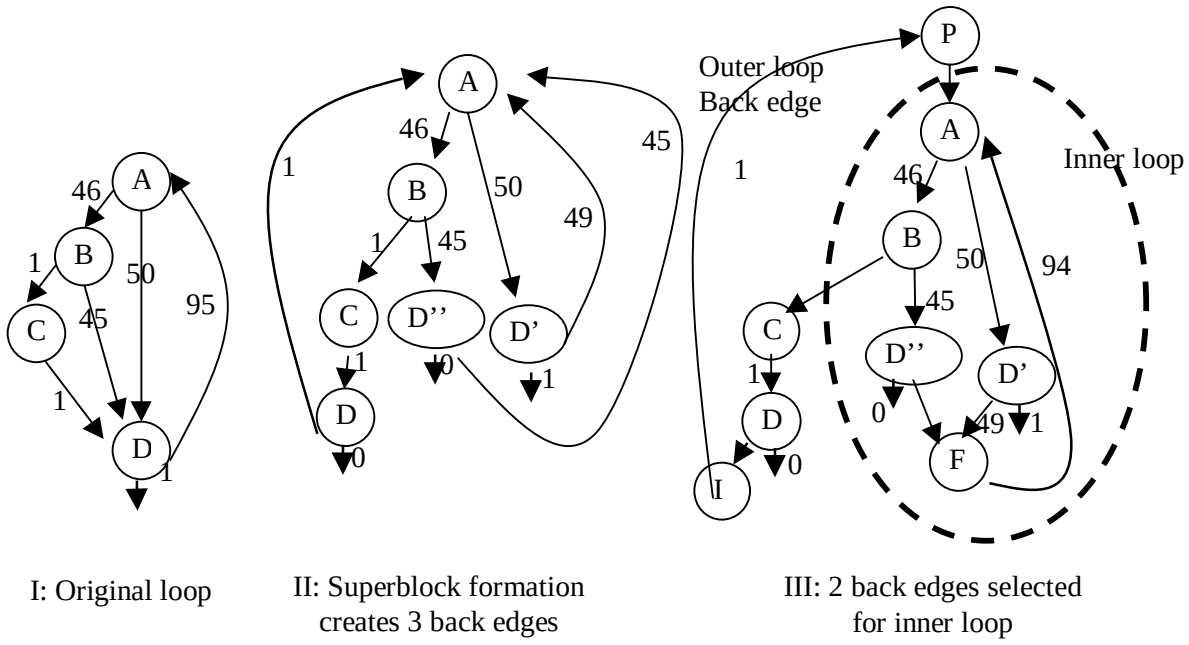

Figure 1: The original program in part I is transformed by the tracer (Section 3.2) to produce the superblocks in part II. The program in part II is transformed by commando loop optimization (Section 3.3) into the program in part III. Flow-graph edges are annotated with execution counts.

Loops entrances and exits terminate a trace for loops with an average iteration count of three or higher. Most trace pickers terminate traces at loop boundaries [13], [15]. However, if the average iteration count is 1 or 2, our trace picker will select a path that enters the loop, includes the loop body, and continues past the loop exit.

Superblock formation is used to change the trace into a single superblock. For traces that do not contain loops, our algorithm is the same as Hwu [15]. We visit the blocks on the trace in trace order, starting with the second block. If a block has more than one predecessor, a copy of the block and its outgoing edges are made and the on-trace incoming edge is redirected to the copy. The execution count of the copied nodes and outgoing edges are scaled by $e_t/\Sigma e_i$ where e_t is the execution count of the incoming on-trace edge and Σe_i is the sum of the incoming edges. The original node and its outgoing edges are scaled by 1- $e_t/\Sigma e_i$. This is repeated until every basic block on the trace is visited. For example, in Figure 1, part I, the trace consisting of the sequence of basic blocks AD is selected, which is the most frequently executed path. The result after super-

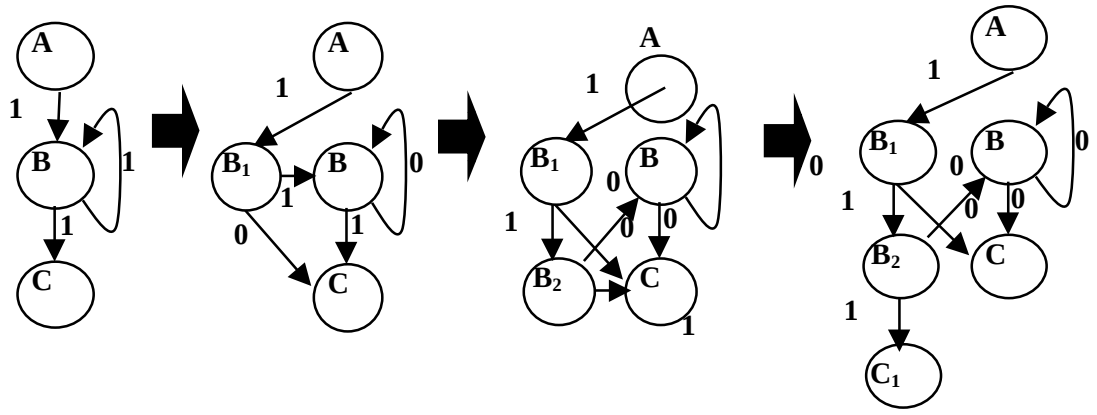

Figure 2: Loop peeling as performed by superblock formation

block formation is shown in part II. The optimization in part III is described in a later section. In the trace AD, basic block D has two predecessors, so it is copied and the on-trace edge (A,D) is redirected to the copy (D'). If the trace had more basic blocks, everything after that point would be copied, too. The result is the superblock AD', which is now only entered from the top. The next trace chosen is BD, and since C also targets D, a new node D'' is created. The (B,D) edge is moved to (B,D''), resulting in superblocks BD'' and CD.

Loops with an average trip count of less than 3 are candidates for loop peeling. For these loops, the trace may extend across the loop entrance and loop exit. Superblock formation for a trace that enters or leaves a loop is handled slightly differently when visiting a basic block that has an outgoing loop back edge. When the loop bottom is visited by the tracer and the back edge has a higher execution count than other successor edges, a decision is made to peel an iteration. The basic block and its outgoing edges are copied. The copied back edge's count is set to e_t and the other outgoing edge, if any, is set to 0. The original loop back edge is reduced by e_t and the original other edge is unchanged. The next block that the tracer visits is the target of the loop back edge, which is the loop top. The tracer continues to visit basic blocks in the trace and will reach the loop back edge again. Since the original back edge count is reduced every time the tracer traverses it, the count will eventually go to 0. If the loop back edge does not have the high-

est execution count, the other edge is followed, usually leading out of the loop. As a failsafe for unusually structured loops, the tracer never traverses an edge twice. The tracer can potentially form a single superblock that contains code before the loop, one or two copies of the loop body, followed by code after the loop.

Figure 2 shows an example of peeling a loop that is a single basic block. B is copied to make B_1. The next block on the trace is B. B is copied again to make B_2. The next block on the trace is C. C is copied to make C_1. The superblock is the path A B_1 B_2 C_1.

Combining loop peeling and superblock formation eliminates a phase ordering problem. If peeling is done first, as is done by Hwu [15], then the whole body of the loop must be copied, not just the important path. If peeling is done after superblock formation, the peeled loop bottom terminates a superblock.

3.3. Commando Loop Optimization

The commando loop optimization is a restructuring transformation that splits a singly nested loop into a doubly nested loop. The frequently executed paths are kept in the inner loop, while the rest are moved to the outer loop. Moving some paths to the outer loop creates opportunities for classical optimizations. For example, a computation may be loop invariant in the restructured inner loop, but not in the original singly nested loop.

The name commando loop optimization comes from some compiler folklore. Loops with many back edges are called commando loops in the GEM backend. This term came from a test program that processed commands in a loop, where every case had its own back edge. The program was written with Portuguese language identifiers and the Portuguese word for command is commando, hence the name commando loop.

After the tracer has processed a loop body, the resulting loop often has multiple back edges. These edges occur when the tracer decides to copy a basic block that contains a back edge, but

can also occur when *continue* statements are used in C programs. This is illustrated in Figure 1, where the superblock transformation is applied to the loop in part I to give the loop in part II. Two copies are made of the block D creating superblocks AD', BD'', and CD, each with their own back edge. When profile information is present, the compiler inserts two new nodes, I and F. The frequently executed back edges are redirected to F and the rest of the back edges are redirected to I as shown in part III. The frequently executed back edges are selected by sorting all back edges by execution count and then including enough of the highest count edges to comprise 90% of the total count for all back edges. Next, the loop is restructured into a doubly nested loop by inserting a preheader (node P), creating a back edge from I to P, and creating a back edge from F to the original loop top.

After commando, there is an inner loop containing the frequently executed paths and an outer loop with the less frequent ones. Applying this transformation alone will not make the program faster; however, other optimizations are now possible. If the infrequently executed paths moved to the outer loop contain procedure calls or stores, there are more opportunities for finding loop invariant expressions, holding values in registers in the inner loop, etc.

This optimization is a generalization of the superblock loop optimizations from Chang [12]. After superblock formation, the most common path through the loop is a superblock loop, which is a superblock with an edge from the bottom to the top. Chang describes specialized forms of loop invariant code removal, global variable migration, and loop induction variable elimination that apply only to superblock loops. In contrast, we restructure the loop so that conventional optimizations, such as loop invariant removal achieve the same effect. Commando can create inner loops with multiple paths; this is beneficial when there are both multiple important paths through the loop and some infrequent paths that can be eliminated. An earlier version of commando only included the most frequent path in the inner loop, in effect creating a superblock loop. We found that this performed poorly in some cases because early exits from the superblock are also early

exits from the loop, which drastically reduces the trip count when the same path is not taken through the loop repeatedly.

3.4. Live-on-Exit Renamer

The live-on-exit renamer was adapted from the Multiflow compiler [13]; it tries to remove a constraint that forces the compiler to create long dependent chains of operations in unrolled loops. After unrolling, dependencies between uses and updates of scalar variables may prevent the scheduler from overlapping iterations. A common technique to solve this problem is to create multiple instances of the variable so that each copy of the loop body operates on its own copy of the scalar. This form of renaming cannot be done when there are multiple exits from the loop and the scalar is live on exit. The reference outside of the loop expects to find the scalar in a single place, forcing every copy of the scalar to use the same storage location.

An illustration of this problem is in Figure 3. The original loop is unrolled by 3. In the unrolled loop, the increment of `i` in line 4 cannot speculate above the branch in line 3 because `i` is live at the target of the branch. If statement 4 stored the result in a place other than `i`, then it could move above the branch. This is not possible because every path that reaches line 8 must put the variable `i` in the same place.

We solve this problem by introducing a unique landing pad at each loop exit that copies the renamed instance of the live variable to the proper location, as is shown in Figure 3. Now the loop index variable increments can be moved above the branch and the loop iterations can be completely overlapped. This is implemented as a self assignment on the landing pad (`i = i;`), which gives the register allocator the freedom to put every lifetime of `i` in a different register.

The commando loop optimization creates loops with early exits, creating opportunities for the live-on-exit renamer. Profile information is used to select the candidates for live-on-exit re-

naming, determining where the improvement in scheduling is likely to outweigh the cost of the extra copy operations.

3.5. Rarely-Called

The rarely-called optimization tailors the register linkage of a procedure so that calls to rarely executed routines preserve the values held in scratch registers. This tailoring allows the caller to keep values in registers that are live across infrequently executed paths that contain calls. The

Original program:

```
while (a[i] != key) i = i+1;
return i;
```

After being unrolled by 3:

```
1. while (a[i] != key) {
2.   i=i+1;
3.   if (a[i] == key) goto E;
4.   i=i+1;
5.   if (a[i] == key) goto E;
6.   i=i+1;
7. }
8. E: return i;
```

After LOE:

```
while (a[i] != key) {
  i1 = i+1;
  if (a[i1] == key) goto E1;
  i2 = i+2;
  if (a[i2] == key) goto E2;
  i = i+3;
}
E: return i;
E1: i = i1; goto E;
E2: i = i2; goto E;
```

Figure 3: Live-on-exit renaming example

optimization has an effect similar to shrink wrapping [16].

The rarely-called optimization interacts with the interprocedural register allocator, which works as follows. Procedures in a single compilation unit are compiled in a bottom up walk of the call graph. Call signatures are recorded for each routine, which includes the registers that are potentially modified by a call to a routine.

When the compiler needs to keep a value live across a call, it may keep the value on the stack, in a callee-saved register, on in a caller-saved register. Each choice has a cost and constraints. There are an unlimited number of stack locations, but using the stack usually requires more memory operations than using a register. The callee-saved registers are very limited in number and have an opening cost because they must be saved and restored by the callee. There are more caller-saved registers and they do not have an opening cost, but to use one the compiler must know that the call does not modify the register. The best allocation choice minimizes the total number of memory operations.

The rarely-called optimization identifies infrequently called routines that may be called from frequently executed routines. The infrequently called routine is given a register signature that forces it to preserve the caller-saved registers. If it does use a caller-saved register, it must restore its value the same way it preserves the value of callee-saved registers. Calls to the infrequently executed routine are now more expensive because of the extra saves and restores, but a caller of the routine now has a large set of registers that it can use to hold values live across paths that contain potential calls to that routine.

```
NODE ***xlsave(NODE **nptr,...)
{
  va_list pvar;
  NODE ***oldstk;
  oldstk = xlstack;
  va_start(pvar,nptr);
  for (;
       nptr != (NODE **) NULL;
       nptr=va_arg(pvar,NODE **)) {
    if (xlstack <= xlstkbase)
      xlabort("stackoverflow");
    *--xlstack = nptr;
    *nptr = NIL;
  }
  va_end(pvar);
  return (oldstk);
}
```

Figure 4: Function from xlisp where rarely-called is beneficial

An example where this optimization is useful is in Figure 4. In the `xlsave` function, the compiler would like to keep the `nptr` variable in a register. The value must be live across the call to `xlabort`, so it cannot normally be put in a scratch register. Using a callee saved requires a save and restore in the procedure prolog/epilog. This is relatively expensive because the loop has an average trip count of less than two. The function `xlabort` is never called in the training run, so the compiler gives it a signature that requires it to preserve all the caller saved registers. Now the `xlsave` function can use all of the caller saved registers to hold values live for the entire loop. Shrink wrapping achieves the same effect by putting saves/restores around the call site and is more flexible. However, the rarely-called optimization was able to piggyback on the pre-existing mechanism for specifying procedure call signatures.

This loop looks like a good candidate for commando loop optimization to move the call to `xlabort` to the outer loop. This does not help very much because of the short trip count, which means that the outer loop is executed almost as frequently as the inner loop. After commando, the compiler has the same problem of keeping the values in registers across potential calls to `xlabort` in the outer loop.

3.6. Register Allocator

The register allocator uses a bin packing technique to allocate variables to registers or stack locations [8], [3], [21]. Without profile information, it estimates relative execution counts by using loop depth. When profile information is available, it uses actual execution counts.

Execution count information is used two ways. First, it is used to model the cost of a particular assignment of registers. The execution count of the basic block that contains a load or store determines its cost.

Second, in frequently executed code, the register allocator adjusts the assignment policy to give the scheduler more freedom to move operations. When the allocator reuses a register for a new lifetime, an antidependence is created between the uses of the old lifetime and the definition in the new lifetime. The antidependence limits the ability of the scheduler to reorder operations. This is the standard phase ordering problem between register allocation and scheduling.

For frequently executed code, the allocator will delay reusing a register for a new lifetime, decreasing the number of antidependencies between instructions that are close together. The drawback to this policy is that the procedure uses more registers, interfering with the interprocedural register allocation. We only apply this optimization to frequently executed code.

We believe that the first use, accurately estimating the cost of register assignments, is the most important, especially for integer programs. The second use, delaying register reuse, is more important for floating point programs where scheduling for long operation latency is important.

3.7. Code Layout

Our code layout algorithm is essentially the same as Pettis and Hansen [4], [5], [6], [9]. Its goal is to reduce instruction cache misses and improve instruction fetch by using profile information to guide the layout of code in memory. We found that the algorithm worked well, except in its handling of branches for programs with very large text sections.

The displacement of a branch instruction in Alpha has a range of plus or minus 4 megabytes. If a branch target is out of range, a longer code sequence must be used. The Pettis and Hansen algorithm places branches and their targets close together if the profile shows that the branch is executed. When the branch is never executed, they can be placed very far apart, overflowing the branch displacement. On large programs this proved to be a serious problem; one binary (100Mbytes) grew by 20% when code layout transformed many branches into long branch sequences. We modified the original algorithm to give priority to branches that are executed, but to never place a branch and its target more than 4 megabytes apart, if possible.

3.8. Miscellaneous

Profile information is used in various other places in the compiler. We describe some examples in this section.

The loop unroller does not unroll loops that never execute and is more willing to unroll large loops if they are frequently executed.

Switch statements in C are implemented with a combination of tests and conditional branches and computed branches through a jump table. With profile information, the compiler inserts a test for the most frequently occurring case before any other tests or jump table lookups [12].

When evaluating *&&* and / / expressions in C, the compiler can swap the operands if a different evaluation order is likely to skip some operations. For example, if the profile indicates that the second operand of a && is usually evaluated, then it is known that the first operand is usually true. If the operands are swapped, then it is possible that only the first operand needs to be evaluated if it is false. The swapping of operands can only be done if it is known that the evaluation has no side effects.

Program	Code Size	Std Dev	Description
GO	399.k	0%	strategy game
M88KSIM	244.k	0%	88K simulator
GCC	1403.k	0%	compiler
COMPRESS	111.k	1%	file compression
LI	188.k	0%	lisp interpreter
IJPEG	258.k	0%	image compression
PERL	436.k	1%	perl interpreter
VORTEX	654.k	1%	database

Table 1: Characteristics of benchmark programs

Padding NOPs are inserted to align instruction for better branch prediction and instruction fetch [17]. When profile information is available, code that is not executed is not padded, and the compiler can also eliminate some pads if the likely branch direction is known.

The only direct use of profile information in the scheduler is to prevent it from speculating operations above likely taken branches. However, the tracer creates large extended basic blocks and aligns the control flow so that scheduling an extended basic block is effective.

4. Evaluation

The system used for evaluation is a Compaq DS20 with a 500MHZ 21264. For benchmarks, we use the programs from SPECInt95 [7]. Some characteristics of the benchmarks are listed in Table 1. As an indication of the run to run variation, the column "Std Dev" is the standard deviation for 9 runs of the baseline configuration, as a percentage of mean run time. The run to run variation is 1% or less, which is small when compared to the effects of optimization.

We use SPECInt95 because they are easy to compile and measure. From our experience with commercial applications, we believe that the large programs in SPECInt95, such as VORTEX and GCC, behave more like real production applications.

The baseline that we compared profile-based optimization against is compiled with the options "`-fast –O4 –inline speed –ifo –assume whole_program -arch ev56`

`-spike`." This is the set of switches that is generally recommended for aggressive optimization [22]. The first three switches turn on aggressive classical optimization. The compiler does whole program optimization with "`-ifo -assume whole_program`". The switch "`-arch ev56`" lets the compiler generate byte and word memory instructions, which is an extension to the original architecture. The switch "`-spike`" turns on post link optimization. For profile-based optimization, we `add` "`-feedback`". Training is done with the SPEC *train* workload, and the benchmarks are timed running the *ref* workload. Profiles were generated with the instrumentation tool pixie. Optimization was disabled for the compile of the training run, which usually gives the best results [23]. For all of our time measurements, we run 9 times and select the median.

In Table 2, we list speedup by optimization and Table 3 measures the code growth. Table 4 is a key to the optimization names. A positive number in Table 2 indicates that the program runs faster when the optimization is turned on. A positive number in Table 3 indicates that the code grew larger.

PROGRAM	ALL	CMNDO	LYOUT	TRCER	RARE	INLINE	LOE	LU	REG	SW	RF	ALIGN
GO	-1%	0%	5%	0%	1%	1%	0%	1%	0%	1%	0%	1%
M88KSIM	59%	5%	0%	0%	2%	45%	2%	0%	2%	2%	0%	0%
GCC	14%	-1%	7%	0%	0%	0%	0%	0%	1%	1%	0%	0%
COMPRESS	4%	0%	0%	1%	-1%	7%	-1%	-1%	1%	0%	-1%	-1%
LI	17%	1%	3%	10%	3%	10%	1%	0%	1%	4%	1%	0%
IJPEG	5%	6%	0%	4%	1%	4%	0%	5%	2%	1%	2%	2%
PERL	10%	0%	-1%	4%	-3%	4%	1%	3%	-1%	6%	1%	0%
VORTEX	36%	0%	20%	2%	-1%	17%	-1%	0%	-1%	0%	0%	1%
SPECINT	17%	1%	4%	3%	0%	10%	0%	1%	1%	2%	0%	0%

Table 2: Speedup over baseline for profile-based optimization.

PROGRAM	ALL	CMNDO	LYOUT	TRCER	RARE	INLINE	LOE	LU	REG	SW	RF	ALIGN
GO	3%	1%	1%	4%	0%	-2%	0%	0%	2%	0%	0%	-6%
M88KSIM	-7%	0%	-2%	2%	1%	4%	0%	-1%	1%	0%	0%	-14%
GCC	-16%	1%	-6%	3%	1%	-3%	0%	-1%	1%	0%	0%	-8%
COMPRESS	-13%	0%	-1%	-3%	0%	-3%	0%	0%	-4%	0%	0%	-14%
LI	-15%	0%	1%	1%	0%	-1%	0%	0%	1%	0%	0%	-18%
IJPEG	-14%	1%	-1%	2%	0%	-1%	0%	-2%	0%	0%	0%	-10%
PERL	-16%	0%	-1%	0%	3%	-5%	0%	0%	1%	0%	0%	-11%
VORTEX	0%	0%	-4%	6%	0%	1%	0%	0%	2%	0%	0%	-7%
AVERAGE	-10%	0%	-2%	2%	1%	-1%	0%	-1%	0%	0%	0%	-11%

Table 3: Code growth over baseline for profile-based optimization.

ALL	All profile-based opt	**LOE**	Live-on-exit renamer, Section 3.4
CMNDO	Commando loop optimization, Section 3.3	**LU**	Allow loop unroller to use profiles, Section 3.8
LYOUT	Code layout, Section 3.7	**REG**	Register allocator use profiles, Section 3.6
TRCER	Superblock formation, Section 3.2	**SW**	Switch statement optimization, Section 3.8
RARE	Rarely-called, Section 3.5	**RF**	Optimize evaluation of &&,||, Section 3.8
INLINE	Allow inliner to use profiles, Section 3.1	**ALIGN**	Use profiles in code alignment, Section 3.8

Table 4: Key to optimization names.

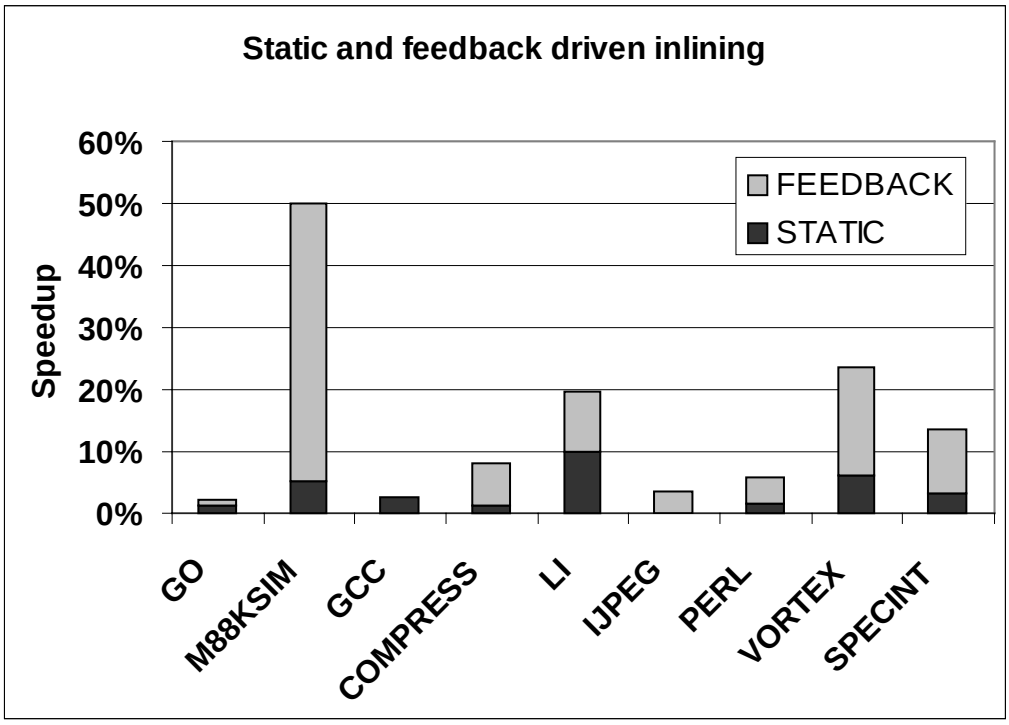

Figure 5: Static and feedback driven inlining

4.1. Speedup

The column ALL is the effect from using all the profile-based optimizations compared against the baseline. In the other columns, we measure the contribution of a single optimization. For INLINE, LU, ALIGN, and REG, we measure the speedup of using profile-based optimization over the conventional static inliner, loop unroller, instruction alignment and register allocator, respectively.

To perform these measurements, we build and run the program with all of the optimizations except the one we are measuring, and then we compute the ratio of ALL versus the current run. We use this method because some of the optimizations rely on other optimizations, and we need to measure them in a context where everything else is turned on. Many of the optimizations interact, so the sum of the individual contributions does not necessarily equal the ALL column. For a

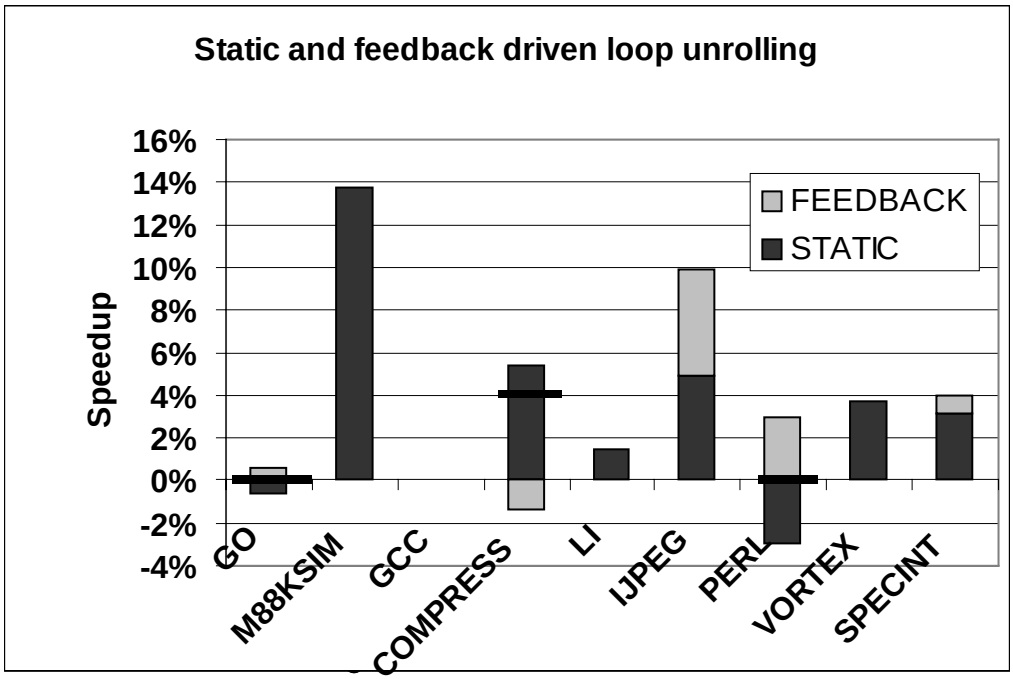

Figure 6: Static and feedback driven loop unrolling

characterization of average speedup, we use the improvement in the SPEC ratio, which is a geometric mean of the per program SPEC ratios.

Almost all of the programs are significantly faster when profile-based optimization is used, one by as much as 59%. The SPECInt ratio is improved by 17%.

The most dramatic speedup comes from the profile-based heuristics for the inliner. In M88KSIM, profile-based inlining gives a 45% speedup over static inlining. Most of it comes from inlining a single routine, `alignd`, where a number of arguments are passed as pointers. Inlining allows the compiler to discover that the arguments point to unaliased local variables, and the compiler is able to keep them in registers. Even if the static inliner were to inline this routine, the profile-based inliner would still contribute 12%, which is comparable to the speedups that the profile-based inliner gives on other programs. From our experience with other programs, we know that a major benefit from inlining comes from eliminating memory operations, both from

solving an aliasing problem, as in M88KSIM, or by eliminating saves and restores in procedure prologs and epilogs.

The commando loop optimization does well for IJPEG and M88KSIM, which spend much of their time in loops with branches [9]. COMPRESS is similar, but does not benefit because the paths through the loop are similarly weighted so the compiler is not able to move any paths to the outer loop. When we tried making the optimization pick only the most frequently executed path for the inner loop, the generated code was worse. Two loads that were common subexpressions in the original flow graph were split between the inner and outer loop, which made it impractical for the compiler to recognize that they computed the same value. Aggressive use of commando requires that the compiler do a very good job of optimizing outer loops.

The effectiveness of code layout correlates well with total code size, except for PERL and M88KSIM. For PERL, we believe unrepresentative training data is a problem. M88KSIM appears to fit in the instruction cache. Commercial applications tend to be very large, and almost always benefit from code layout.

The tracer does well on programs with complicated control flow, although we expected more improvement for GCC. The switch statement optimization helps LI and PERL, which do spend time in functions that have switch statements. GCC also spends time in functions with switch statements, but profiles show that there usually is not a single dominant case for the switch. A more sophisticated switch optimization may be needed here.

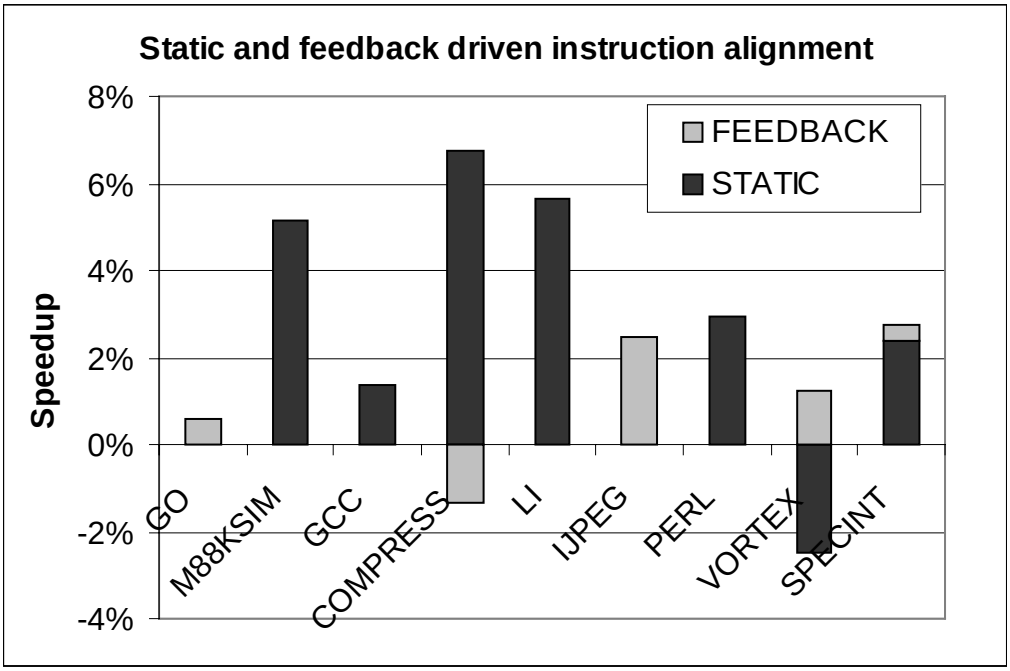

Figure 7: Static and feedback driven instruction alignment

The rest of the optimizations had smaller effects on performance. However, it is possible that our methodology understates the benefit of some optimizations. If two optimizations achieve similar effects, then turning off one optimization will not affect performance because the other optimization will still be on. Turning off both at the same time would show that at least one of them is needed to achieve the performance of the "ALL" configuration. For example, GCC achieves a 14% speedup, but the sum of the individual optimizations is only 9%, so we are probably undercounting some optimization. A more sophisticated analysis of the interactions might identify the source of the speedup [24].

4.2. Code Growth

Code growth is broken down by optimization in Table 3. Total code size growth is not necessarily a good indicator for instruction cache effects, which are not directly measured in this paper. However, negative effects on instruction cache hit ratio are reflected in the actual runtime.

Code size is usually lower with PBO because code expanding optimizations are not applied to parts of the program that are not executed. Most of the reduction comes from inserting less padding for instruction alignment, but profile information helps to reduce the code growth from loop unrolling and inlining as well. Code layout reduces the number of taken branches, which further reduces the need for padding.

The tracer is the only optimization that makes code grow significantly on average. It does not optimize code that is rarely executed, typically ignoring 90% of large programs. Thus, a few percent growth in code for the tracer can be a very large increase in instruction cache footprint. Since the larger code is not always rewarded with better performance in the tracer, we probably need to tune this optimization.

Although the profile-based inliner makes most programs smaller, it does make M88KSIM grow. Profiles show that after inlining 60% of the time is spent in a single routine and instructions cache misses are low. Considering the large speedup, the code growth is well worth it.

4.3. Comparison of Static and Profile-Based Heuristics

Many of the optimizations are driven by a cost model of profitability, and profile information is just one factor considered. For some of those optimizations, we measured the relative contribution using the static and profile-based cost models. For the baseline in these graphs, we turn off the static and profile-based optimization, where the previous measurements used full static optimization as a baseline. When optimizations have a negative effect on performance, a bar indicates the net speedup when both are used.

Performance of the inliner is shown in Figure 5. The static inliner does improve performance, but most of the benefit comes when profile information is available.

As shown in Figure 6, most of the gain in loop unrolling comes from the static unroller, and a small additional benefit comes from profile information, on average. PERL and IJPEG do get a significant boost from profile directed unrolling.

Performance for code alignment can be found in Figure 7. Static code alignment improves performance but greatly increases the code size, and the feedback driven version achieves slightly better performance with much smaller code size.

4.4. Beyond SPEC

This study only uses the SPEC benchmarks, but Compaq has several years experience applying profile-based optimization to a wide variety of production software [9], [19]. Compaq and its ISV's have shipped databases, compilers, CAD programs, device drivers, and graphics software that have been built with profile-based optimization. It is not practical to repeat the experiments in this paper for these products because it requires access to source code to build the various configurations. However, Cohn [9] reports that using code layout results in a 20% speedup for the compiler, a 10% speedup for transaction processing, and a 32% speedup for AutoCad, a drawing program. Ayers [19] describes using whole program profile-based optimization on a CAD program for the HP PA. Schmidt [18] reports a 7% speedup for transaction processing on IBM's AS400.

Compared to the benchmarks used in this paper, we have found that real applications are more likely to benefit from profile-based optimization and usually have greater speedups. We believe this happens because the SPECInt95 code and data sizes are small [9] and tend to fit in the on-chip caches.

5. Summary and Conclusions

Profile-based optimization has been integrated into the GEM compiler. Rather than develop a totally new set of optimizations, we added a few profile-based restructuring transformations and

leveraged our very strong conventional optimizer. Existing optimizations were adapted to consider execution counts as part of the cost model for decision. The amount of code devoted specifically to profile-based optimization is very small compared to the total amount of code in the compiler. It is about 1%, most of it being in the system to manage profiles.

Profile-based optimizations provide impressive speedups over our static heuristics. The largest speedups came from inlining.

Profile-based optimization has been deployed for many of the performance sensitive programs running on Alpha. A huge effort goes into tuning every part of the system to maximize performance [20], and profile-based optimization is just one part of it. The main obstacle that we have found is ease of use. Generating profiles requires additional steps, which leads to opportunities for user error. Compaq has worked on making the process as transparent as possible [23], [9], [14], and we expect profile-based optimization to be even more widely used in the future.

Acknowledgements

Gene Albert developed the system for managing profiles, wrote new optimizations and improved others. Michael Adler, David Blickstein, Peter Craig, Caroline Davidson, Neil Faiman, Kent Glossop, David Goodwin, Rich Grove, Lucy Hamnett, Steve Hobbs, Bob Nix, Bill Noyce, and John Pieper contributed ideas and developed profile-based optimizations.

References

[1] H. Rotithor, K. Harris, and M. Davis**, "**Measurement and Analysis of C and C++ Performance," Digital Technical Journal, vol. 10, no. 1 (1999): 32-47

[2] L. Wilson, C. Neth, and M. Rickabaugh**, "**Delivering Binary Object Modification Tools for Program Analysis and Optimization," Digital Technical Journal, vol. 8, no. 1 (1996): 18-31.

[3] R. Johnson, "An Approach to Global Register Allocation," Ph.D. thesis, Carnegie-Mellon University, December 1975.

[4] W. Hwu and P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," Proceedings of the Sixteenth Annual International Symposium on Computer Architecture, Jerusalem, Israel (June 1989).

[5] K. Pettis and R. Hansen, "Profile Guided Code Positioning," Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, N.Y. (June 1990): 16-27.

[6] S. McFarling, "Program Optimization for Instruction Caches," ASPLOS III Proceedings, Boston, Mass. (April 1989): 183-193.
[7] Information about the SPEC benchmarks is available from the Standard Performance Evaluation Corporation at http://www.specbench.org/.
[8] D.S. Blickstein, P.W. Craig, C.S. Davidson, R.N. Faiman, K.D. Glossop, R.P. Grove, S.O. Hobbs, and W.B. Noyce. "The GEM Optimizing Compiler System," Digital Technical Journal, vol. 4, no. 4 (1992): 121-136.
[9] R. Cohn, D. Goodwin, and P.G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike," Digital Technical Journal, vol. 9, no. 4 (1997): 3-20.
[10] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools (Reading, Mass.: Addison-Wesley, 1985).
[11] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, C-30, 7 (July 1981): 478-490.
[12] P. Chang, S. Mahlke, and W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," Software-Practice and Experience, vol. 21, no. 12 (1991): 1301-1321.
[13] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. O'Donnell, and J. C.., "The Multiflow Trace Scheduling Compiler," The Journal of Supercomputing, vol. 7, no. 1/2 (1993): 51-142.
[14] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?" Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint-Malo, France (October 1997): 1-14.
[15] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation ," The Journal of Supercomputing, Kluwer Academic Publishers, 1993, pp. 229-248.
[16] F. Chow, "Minimizing register usage penalty at procedure calls," In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 85-94, Atlanta, Georgia, 22-24 June 1988. *SIGPLAN Notices* 23(7), July 1988.
[17] "Compiler Writer's Guide for the Alpha 21264," http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html
[18] W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky, "Profile-directed restructuring of operating system code," IBM Systems Journal 37, No. 2 (1998): 270-297
[19] A. Ayers, et al., "Scalable Cross-Module Optimization," In Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, Montreal, Canada (June 1998): 301-312.
[20] T. Kawaf, D. J. Shakshober, and D. C. Stanley., "Performance Analysis Using Very Large Memory on the 64-bit AlphaServer System," Digital Technical Journal, vol. 8, no. 3 (1996): 58-65
[21] O. Traub, G. Holloway, and M. Smith. "Quality and Speed in Linear-scan Register Allocation," *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 142-151, June 1998
[22] Information about compiler switches can be found at: http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V50_HTML/REFPSHLF.HTM
[23] G. Albert, "A Transparent Method for Correlating Profiles with Source Programs," Proceedings of the Second Workshop on Feedback Directed Optimization, November, 1999.
[24] K. Chow and Y. Wu, "Feedback-Directed Selection and Characterization of Compiler Optimizations," Proceedings of the Second Workshop on Feedback Directed Optimization, November, 1999.