To appear in: Proceedings of MICRO29, December 2, 1996

Hot Cold Optimization of Large Windows/NT Applications

Robert Cohn (rc@vssad.hlo.dec.com)
P. Geoffrey Lowney (lowney@vssad.hlo.dec.com)
Digital Equipment Corporation
Hudson, Massachusetts

Abstract

A dynamic instruction trace often contains many unnecessary instructions that are required only by the unexecuted portion of the program. Hot-cold optimization (HCO) is a technique that realizes this performance opportunity. HCO uses profile information to partition each routine into frequently executed (hot) and infrequently executed (cold) parts. Unnecessary operations in the hot portion are removed, and compensation code is added on transitions from hot to cold as needed. We evaluate HCO on a collection of large Windows NT applications. HCO is most effective on the programs that are call intensive and have flat profiles, providing a 3-8% reduction in path length beyond conventional optimization.

1. Introduction

Typically, compiler writers study code listings to look for optimization opportunities. However, we have found that a dynamic view, studying instruction traces, gives a very different perspective on code quality[Sites95]. High quality code often looks unoptimized when viewed as a dynamic instruction trace.

Below, we list an instruction trace fragment from the Windows NT kernel running on a Digital Alpha [Sites95]. The register `ra` is used to hold the return address when a call is made, `sp` is the stack pointer, `a0` is an argument register, `v0` is the return value, `t0` is a scratch register (caller saved), and `s0-s2` are preserved registers (callee saved).

```
bsr      ra,8009e3f0  ; call
lda      sp,-30(sp)   ; grow the stack
stq      s0,0(sp)     ; save the old value
stq      s1,8(sp)     ;  of some preserved
stq      s2,10(sp)    ;  registers
stq      ra,18(sp)    ;
bis      zero,a0,s0   ; copy a0 to s0
ldl      t0,9b8(gp)   ; load,..
cmpult   t0,#13,v0    ; compare, ...
bne      v0,8009e42   ; branch
bis      zero,zero,v0 ; clear v0
br       zero,8009e4f4 ; branch
ldq      s0,0(sp)     ; restore the old
ldq      s1,8(sp)     ;   value of some
ldq      s2,10(sp)    ;preserved registers
ldq      ra,18(sp)    ; load ret address
lda      sp,30(sp)    ; shrink the stack
ret      zero,(ra)    ; return
```

The trace appears to execute many unnecessary instructions. However, looking at the full function listing, they all are necessary. The return address (`ra`) is saved and restored but never modified. This function contains a call, but this code path does not execute it. The preserved registers `s1` and `s2` are also saved and restored but never used. Somewhere in the function, `s1` and `s2` are modified, but the trace does not go there. The function does modify the preserved register `s0` with a copy, but the value stored in `s0` is never read in the instruction trace so the copy and the save/restore of `s0` are unnecessary. The only truly necessary instructions is the load, compare, branch, copy sequence, which form an excellent inline candidate.

An insight into how an optimizer can take advantage of this type of situation came from a tool called NT OM, that arranges code for instruction cache performance[+]. Using profile information, NT OM lays out code so that the fall-through path is the common path and also splits routines into a frequently executed (hot) part and infrequently executed (cold) part [Pettis90, Calder94, Hwu89, McFarling89]. If there is only one path through a routine, the instruction trace and the hot part are identical. Usually, there are several common paths through the routine, and the hot part is the collection of these paths.

If we could optimize the hot part of the routine, ignoring the cold part, we could eliminate all the unnecessary instructions. We create a "hot" routine by copying the frequently executed basic blocks of a function. All calls to the original routine are redirected to the hot routine. Flow paths in the hot routine that target basic blocks that were

[+] NT OM is Digital's executable optimization technology [Srivastava94,Wilson96] written for NT not copied are redirected to the appropriate basic block in the original "cold" routine; that is, they jump into the middle of the original routine. We then optimize the hot routine, possibly at the expense of the flows that pass through the cold path. The process is called Hot Cold Optimization (HCO).

2. Overview of HCO optimization

We describe HCO in detail later; in this section we work through an example, a simplified version of the function associated with the above trace.

```
1.  foo:      lda sp,16(sp) ; grow stack
2.            stq s0,0(sp)  ; save s0
3.            stq ra,8(sp)  ; save ra
4.            addl a0,1,s0  ; s0 = a0+1
5.            addl a0,a1,a0 ; a0= a0+a1
6.            bne s0,L2     ;brnch if s0!=0
7.      L1:   bsr f1        ; call f1
8.            addl s0,a0,t1 ; t1=a0+s0
9.            stl t1,40(gp) ; store t1
10.     L2:   ldq s0,0(sp)  ; restore s0
11.           ldq ra,8(sp)  ; restore ra
12.           lda sp,-16(sp); pop stack
13.           ret (ra)      ; return
```

Assume that the branch in line 6 is almost always taken and that lines 7-9 are almost never executed. When we copy the hot part of the routine we exclude lines 7-9 as follows:

```
a)      foo2: lda sp,16(sp)
b)            stq s0,0(sp)
c)            stq ra,8(sp)
d)            addl a0,1,s0
e)            addl a0,a1,a0
f)            beq s0,L1
g)            ldq s0,0(sp)
h)            ldq ra,8(sp)
i)            lda sp,-16(sp)
j)            ret (ra)
```

Note that the sense of the branch was reversed and its target was changed to `L1` in the original routine. All calls to foo are redirected to the hot routine `foo2`, including indirect calls. If the branch in line f is taken, then control transfers to line 7, which is in the middle of the original routine. Once control passes to the original routine, it never passes back to the hot routine. This feature of HCO enables optimization; when optimizing the hot routine, we can relax some of the constraints imposed by the cold routine.

So far, we have set up the hot routine for optimization, but have not made it any faster. Now we show how to optimize the routine. The hot routine no longer contains a call; we can delete the save and restore of the return address in lines c and h. If the branch transfers control to `L1` in the cold routine, we must arrange for `ra` to be saved on the stack. In general, whenever we enter the original routine from the hot routine, we must fix up the state to match the expected state. We call the fix-up operations compensation code. To insert compensation code, we create a stub, and redirect the branch in line f to branch to the stub. The stub saves `ra` on the stack and branches to `L1`.

Next, we see that the instruction in line e writes `a0`, but the value of `a0` is never read in the hot routine. However, it is not truly dead because it is still read if the branch in line f is taken. We delete the instruction from the hot routine and place a copy on the stub.

HCO tries to eliminate the uses of preserved registers in a routine. Preserved registers can be more expensive than scratch registers because they must be saved and restored if they are used. Preserved registers are typically used when the lifetime of the value crosses a call. In the hot routine, no lifetime crosses a call and the use of a preserved register is unnecessary. We rename all uses of `s0` in the hot routine to use a free scratch register `t2`. We insert a copy on the stub from `t2` to `s0`. We can now eliminate the save and restore in lines b and g and place the save on the stub.

We have eliminated all references to the stack in the hot routine. The stack adjusts can be deleted from the hot routine and the initial stack adjust placed in the stub. The final code and the stub are listed below. The number of instructions executed in the frequent path has been reduced from 10 to 3. If the stub is taken, then the full 10 instructions and an extra copy and branch are executed.

```
1.      foo2:  addl a0,1,t2
2.             beq t2,stub1
3.             ret (ra)
4.      stub1: lda sp,16(sp)
5.             stq s0,0(sp)
6.             stq ra,8(sp)
7.             addl a0,a1,a0
8.             mov t2,s0
9.             br L1
```

Finally, we would like to inline the hot function. Copies of instructions 1 and 2 can be placed inline. For the inlined branch, we must create a new stub that materializes the return address into `ra` before transferring control to `stub1`. Except for partial inlining, we have implemented all of these optimizations in our system.

In the following section, we present the NT applications we used to evaluate HCO and discuss the results. We then present the details of our method. We close the paper with a discussion of related work, and our conclusions.

3. Characteristics of NT Applications

For our experiments, we use programs that are representative of the types of applications run on high performance personal computers (PC's). We also include some programs from SpecInt95 for comparison, although our discussion in the paper focuses on the PC applications. Table 1 identifies the programs and the workloads used to exercise them. All programs are optimized versions of Digital Alpha binaries and are compiled with the same highly optimizing back end that is used on Unix and VMS[Blickstein92].

For our profile-directed optimization, we use the same input for training and timing so that we can know the limits of our approach. Others have shown that a reasonably chosen training input will give reliable speedups for other input sets [Calder95]. Our experience with program layout optimizations for our benchmark programs confirms this result.

Table 2 lists some static and dynamic characteristics of the single executable or dynamically linked library (DLL) responsible for a majority of the execution time for each application. The smallest PC application is three times larger than the largest SpecInt95 program in our list, and the largest PC application, ACAD, is twenty times larger. All of them have thousands of functions. Call overhead is high and ranges from 7%-16%, except for the more loop intensive TEXIM and MAXEDA. We approximate call overhead by measuring time spent in procedure prologs and epilogs, which is mostly stack adjusts and saving and restoring of preserved registers. The time spent in leaf routines is low, with the exception of WINWORD.

Program	Full Name	Type	Workload
VC	Microsoft Visual C	compiler backend	5000 lines of C code
SQLSERVR	Microsoft Sqlserver 6.5	database	cached TPC-B
ACAD	Autodesk Autocad	mechanical cad	SDUG benchmark
EXCEL	Microsoft Excel 5.0	spreadsheet	BAPCO
USTATION	Bentley Systems Microstation	mechanical cad	rendering
WINWORD	Microsoft Word 6.0	word processing	BAPCO
TEXIM	Welcom Software Texim 2.0	Project management	BAPCO
MAXEDA	OrCad MaxEDA 6.0	electronic cad	BAPCO
VORTEX	SpecInt95	database	SPEC ref
GO	SpecInt95	game	SPEC ref
M88KSIM	SpecInt95	simulator	SPEC ref
LI	SpecInt95	lisp interpreter	SPEC ref
COMPRESS	SpecInt95	compression	SPEC ref
IJPEG	SpecInt95	JPEG	SPEC ref

Table 1: Benchmark programs and their workloads

In Table 2, we list the percent of execution time spent in the top 5 routines for each routine. VC, SQLSERVR, ACAD, and EXCEL tend to have flat profiles, while the other PC applications have a single routine responsible for a large portion of the execution time.

The focus of our work is programs that have flat profiles and are more call intensive than loop intensive. In

Program	Text Size MB (static)	Routines (static)	Call overhead (dynamic)	Leaf routines (dynamic)	% time spent in the 5 most frequently executed routines 1st	2nd	3rd	4th	5th
VC	2.08	2139	11%	8%	10.9	5.5	5.5	2.6	2.5
SQLSERVR	3.11	3213	16%	2%	6.1	5.6	3.6	3.4	3.2
ACAD	9.29	22097	9%	16%	2.8	2.8	2.2	1.8	1.7
EXCEL	6.81	12129	13%	27%	10.4	4.3	3.9	3.7	3.2
USTATION	3.86	13302	12%	15%	34.4	10	9.9	6.2	6
WINWORD	6.97	12230	7%	50%	38.9	6.8	4.7	4.3	4
TEXIM	1.39	1953	2%	13%	77.1	7.4	3.2	1.3	1
MAXEDA	1.73	1807	2%	4%	34.4	10	9.9	6.2	6
VORTEX	.47	820	23%	2%	16.2	12.3	8.6	8.5	5.2
GO	.29	462	5%	30%	19.8	14.4	9.5	5.9	4
M88KSIM	.17	382	12%	43%	27.5	12.8	11.3	7.7	5.9
LI	.12	489	24%	10%	13.4	13.3	9.7	8.8	8.7
COMPRESS	.06	122	1%	28%	25.7	25.9	19.5	9.4	6.4
IJPEG	.18	408	2%	56%	14.5	14.3	13.4	11.6	10.7

Table 2: Static and dynamic characteristics of benchmark programs

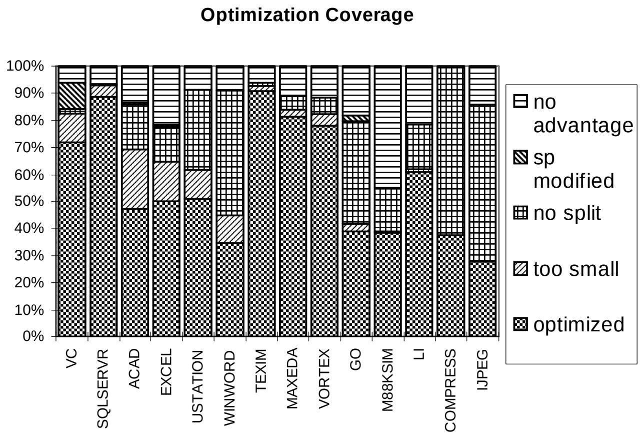

Figure 1: HCO coverage by execution time

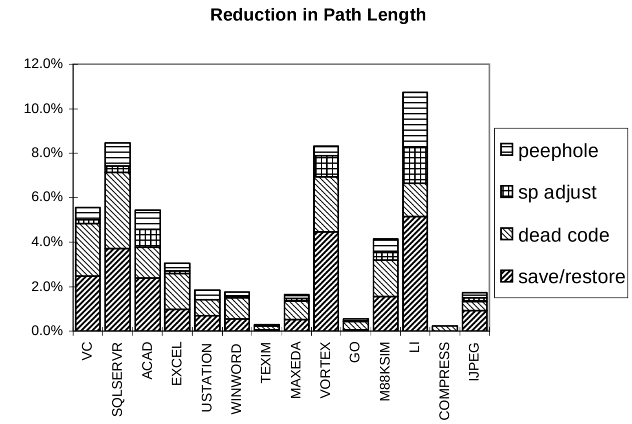

Figure 2: Reduction in path length

these programs, loops typically include function calls and span multiple procedures. The most frequently executed basic block in a function is often the entry point and call overhead tends to be high. Our experience with these programs has shown that optimizations that do not preserve the locality of the instruction stream can adversely affect performance. They all speed up from 5% to 15% when we apply code layout techniques [Pettis90, Calder94, Wilson96].

In this paper, all statistics based on execution time and speedups use path lengths (number of instructions executed) for the dominant executable or DLL. Path lengths are calculated by multiplying the number of instructions in a basic block by the number of times the block is executed. Using path lengths rather than measured times allows us to provide more detailed information about the contribution of each type of optimization. Since it is possible to reduce the path length at the expense of more cache misses, we used simulation and other techniques to verify that instruction cache misses were not affected for LI, VC, ACAD, and SQLSERVR. For the full set of HCO optimizations together, we verified that path length reductions resulted in equivalent run-time speedups for LI, VC, and ACAD. For SQLSERVR, the run-time speedup is less than the path length reduction (8%), and we continue to investigate this discrepancy.

4. Results

We present the results for the full suite of HCO optimizations, except for partial inlining, which has not yet been implemented. The optimizations are partial dead code elimination, which is the removal of code dead in the hot routine; stack pointer adjust elimination, which is the removal of the stack adjusts in the hot routine; preserved register elimination, which is the removal of the save and restore of preserved registers in the hot routine; and peephole optimization, which is the removal in the hot routine of self-assignments and conditional branches with an always-false condition. The optimizations were implemented in NT OM, an optimizer that operates on Alpha NT executables.

In Figure 1, we show coverage statistics for the HCO optimization. Coverage represents the percentage of execution time spent in each category. To compute coverage, we first assign each function to a category, and then for each category sum the execution time of its functions. The category "optimized" is the portion of the execution time that is in functions optimized by HCO. Optimization coverage is typically 60%, but is often higher. The category "too small" is the set of functions where the execution time is so small (< .1% of total time) it did not appear worthwhile to optimize them. Ignoring functions with small execution time allows us to optimize less than 5% of the program text, a significant reduction in optimizer time. The category "no split" represents the functions that we could not split into a hot and cold part because all basic blocks had similar execution counts. The category "sp modified" is for functions where the stack pointer is modified after the initial stack adjust in the prolog. We decided not to optimize these functions, but it is possible to do so with extra analysis. It was infrequent except for VC, where it is 7% of the program and occurs in 2 functions. Finally, the category "no advantage" is for the functions that were split but the optimizer wasn't able to make any faster.

In Figure 2, we show the overall reduction in path length for HCO, broken down by optimization. Most of the reduction in path length comes equally from removal of unnecessary save/restores and partial dead code.

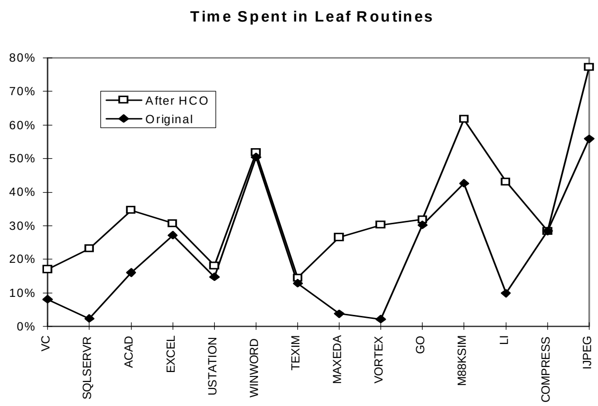

Figure 3: Time spent in leaf routines before and after HCO

Removing stack pointer adjusts and peephole optimizations are a smaller additional gain. When the peephole category is large it is usually because there is a save and restore of a preserved register that is made unnecessary by HCO, and the restore is converted to a self assignment by copy propagation, which is then removed by peephole optimization.

HCO is most effective on call intensive programs such as VC, SQLSERVR, and ACAD, where we eliminate calls when creating the hot routines. For WINWORD, the speedup is small because coverage is low; we could not find a way to split the routines. For EXCEL, HCO was able to split the routines, but there is often a call in the hot path. Inlining may help here, but frequently the call is to a shared library.

HCO is less effective on loop intensive programs such as USTATION, MAXEDA, and TEXIM. HCO provides a framework for optimizing loops, and Chang has shown that eliminating the infrequent paths in loops enables additional optimizations such as loop invariant removal [Chang91]. However, our current implementation has almost no information about the aliasing of memory operations and it can only optimize operations to local stack locations, such as spills of registers.

4.1 Leaf routines

Figure 3 compares the amount of time spent in leaf routines before and after HCO is applied. By eliminating infrequent code, HCO is able to eliminate all calls in functions that represent 10-20% of the execution time in VC, ACAD, SQLSERVR, and MAXEDA. The change in time spent in leaf routines for the other PC applications is very small. Most of the PC applications spend much less than half of the time in leaf routines. Since so much time

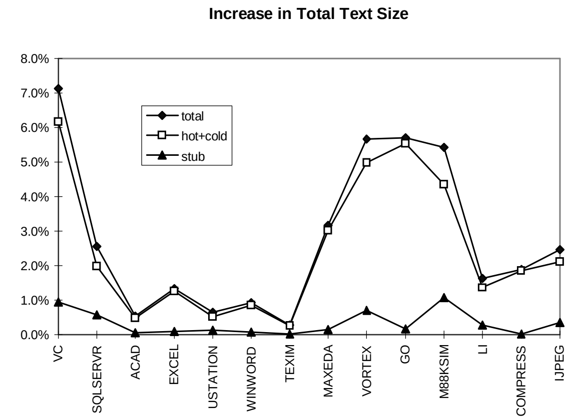

Figure 4: Overall increase in text size

is spent in code with calls in the frequent path, it is important to optimize well in the pre sence of calls.

4.2 Code size

Code size and its effect on cache behavior is a major concern for us. In large applications, locality for instructions is present but not high. If an optimization decreases path length but also decreases locality as a side effect, the net result can be a loss in performance.

Figure 4 shows the total increase in text size from optimization. "Hot+cold" is the part of increase that comes from replacing a single routine with the original routine plus a copy of the hot part. "Stub" is the increase attributed to stub routines. Overall the increase in size is small. The maximum increase is 7.1% for VC. Sqlservr has the best speedup and is only 2.6% bigger. Looking at the increase in total text size is misleading, however. HCO is not applied to routines that are executed infrequently, which typically accounts for more than 95%

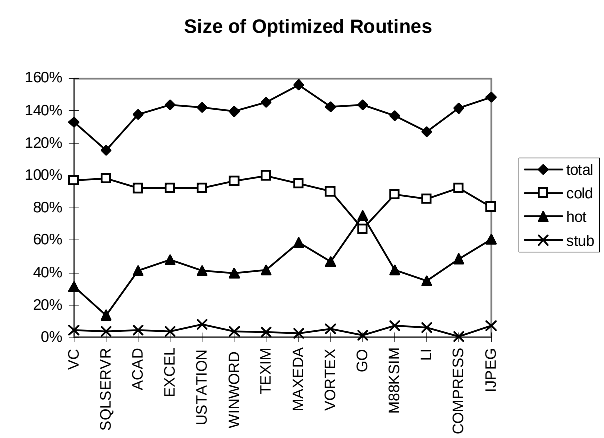

Figure 5: Size of optimized routines

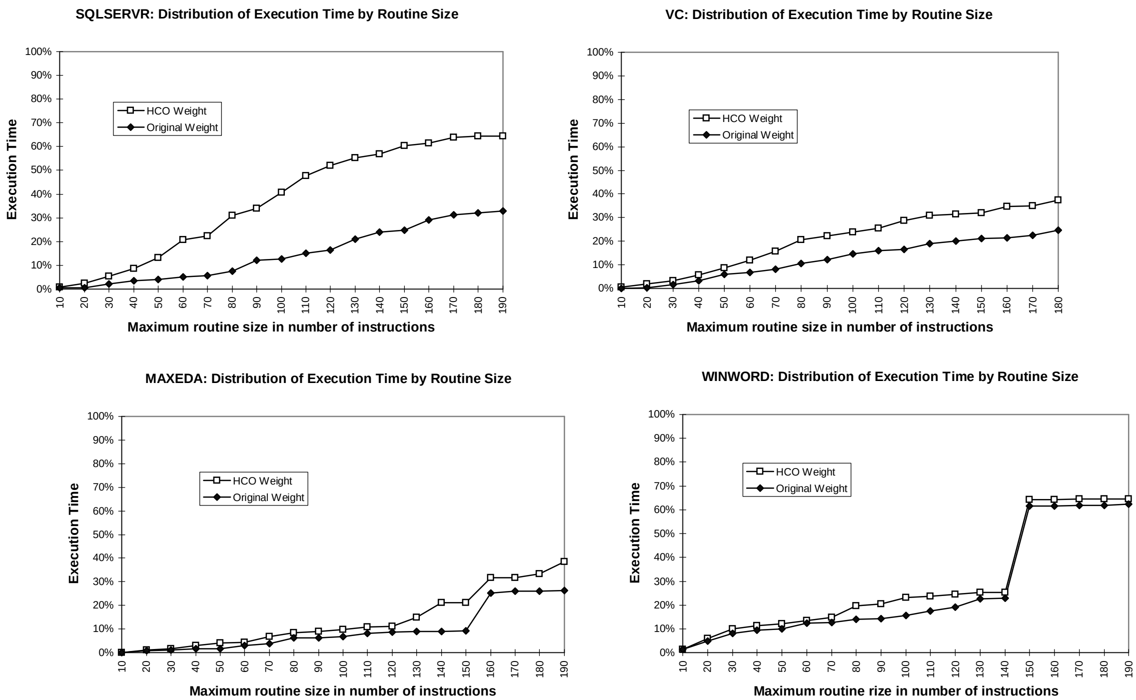

Figure 6: Distribution of routine size and execution time

of the program text, so tripling the size of optimized routines would only result in a modest increase in text size. However, tripling the size of the active part of an application will usually have a disastrous effect on performance.

For this reason, we also measure size increases based on the routines that are optimized. In Figure 5, we compare the total size of all the hot routines to the total size of the original routines they were derived from. By copying just the frequently executed part of the routine, we exclude about 50% of the original routine. Next, we eliminate code that is frequently executed, but is only reachable through an infrequently executed path and is therefore unreachable in the hot routine. This is usually only 1%. Finally, we optimize the hot routine, reducing the remaining code size by about 10% (5% of the size of the original routine). The final size of the hot routines as a percentage of the size of the original routines is shown in the line labeled "hot." By making the most frequently executed part of the program 50-80% smaller one would expect a big improvement in instruction cache behavior. It does; however, it would be misleading to attribute this improvement to HCO since a simpler optimization in NT OM already achieves the same result. As part of the optimization for instruction cache behavior, NT OM splits routines into two parts, one for the frequently executed code and another for infrequently executed code. The frequently executed parts are packed together so that they are less likely to conflict in a direct mapped cache. When HCO is turned on, the cache layout optimizations are run after HCO. The baseline we compare against also has cache optimizations turned on, so improvements attributed to HCO are improvements beyond what the other optimizations can do. HCO does make the frequently executed parts 10% smaller, but we did not see better instruction cache behavior when we ran programs with a cache simulator.

If we were to do partial inlining, only the hot routine would be copied. Since the hot routine is less than half the size of the original routine, this would greatly reduce the growth in code size due to inlining.

The line labeled "cold" in Figure 5 shows how the size of the cold routine is affected by HCO. When we redirect all calls to the hot routine, some of the code in the original routine becomes unreachable. This is usually less than 10%, which is much smaller than the 50% of the code we copied to create the hot routine. Apparently, the infrequent paths in a routine often rejoin the frequent paths, which makes it necessary to have a copy of both in the original routine.

The line labeled “stub” is the code size of the stubs, which is very small. We also implemented a variation of HCO that avoided stubs by re-executing the procedure from the beginning instead of trying to use a stub to fix up the state and jump into the middle. It usually isn’t possible to re-execute the procedure because arguments had already been overwritten. Given the small cost of stubs, we did not pursue this method.

The line labeled “total” shows that HCO makes the total code(hot+cold+stub) 20-50% bigger. A routine is partitioned so that there is less than a 1% chance that the stub and cold part are executed so their size shouldn’t have a significant effect on cache behavior as long as the profile is representative.

Figure 6 shows how splitting affects the distribution of time spent among different routine sizes for two programs where HCO is effective (VC and SQLSERVR), and two programs where it is not (MAXEDA and WINWORD). For each graph, the horizontal axis is the routine size in number of instructions and the vertical axis is the percentage of execution time spent in routines of the maximum size or smaller. The farther apart the two lines, the better HCO was at shifting the distribution from large routines to smaller routines. It is interesting to note that most of the programs spend a large percentage of the time in large functions, which suggests that compilers need to handle complex control flow well, even if profile information is used to eliminate infrequent paths.

5. Optimization

In this section, we describe in more detail how NT OM constructs the hot routine and how it is optimized. HCO was implemented as part of NT OM, a tool for optimizing executables and dynamically loaded libraries (DLLs) for Alpha Windows/NT [Srivastava94, Wilson96, Larus95]. NT OM reads in an executable, identifies the code and data, optimizes the code, and writes out a new executable. Because NT OM can examine the entire executable, it is able to do optimizations that a compiler cannot easily do. For example, NT OM can see the entire call graph, which makes it possible to lay out code to minimize instruction cache misses. However, NT OM does not have some information that is available to a compiler. Currently, NT OM derives the dataflow from the code itself; it does not use extra information from the compiler. The biggest limitation is the lack of memory alias information, which restricts the type of optimizations that can be done on instructions that touch memory.

NT OM can instrument executables to collect profile information in a manner similar to ATOM[Srivastava94b, Wilson96], and uses profile information when optimizing a program, as is done for code layout and HCO.

5.1 Partitioning a routine into hot and cold

Deciding what code to include in the hot routine is a tradeoff between two factors. When we exclude code from the hot routine (especially calls) we create opportunities for optimizations that decrease path length. However, the more code we exclude from the hot routine, the higher the probability that we enter the original routine. The transition from hot to cold is expensive because it is likely to cause cache misses.

We use the probability of exiting the hot routine through a stub to decide how much code to include in the hot routine. The exit probability is the number of times the program exits a hot routine through a stub for each time the routine is called. It is calculated from profile information.

Our experiments showed that path length reductions are not that sensitive to exit probability, but that cache misses and real performance are very sensitive. Doubling the exit probability results in a small improvement in cycles eliminated because of path length reduction but a large increase in additional cycles spent in cache misses. For this reason we tuned HCO to get the biggest reduction in path length that doesn’t cause a significant increase in cache misses. We found that using a very low probability of 1% works best for most programs.

We tuned HCO to avoid cache misses by adjusting the exit probability for maximum measured speedup, rather than path length reduction. For the programs with short execution times, we confirmed that extra cache misses are not a problem through simulation.

5.2 Partially Dead Code

Sometimes, instructions are live on one path and dead on others. This is called partially dead code[Knoop94]. When an operation in the hot part computes a value that is only consumed in the cold part, we optimize this by moving the instruction from the hot routine to some or all of the stubs, which are less frequently executed.

The candidates for partially dead code are found by ignoring the branches to stubs when computing liveness. We do not eliminate dead stores or loads because we only have limited information on aliasing of memory operations.

Next, we find the set of stubs where the definition generated by the dead candidate may reach the stub, and

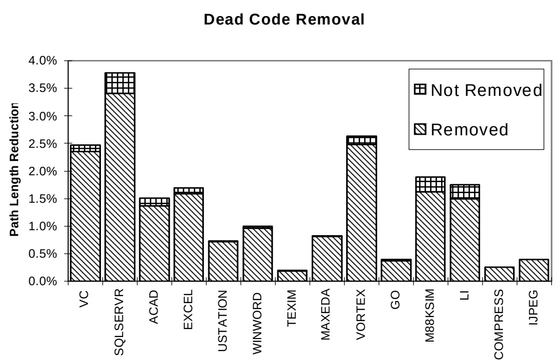

Figure 7: Potential improvements for dead code removal

the register holding the value is live at the stub. If the definition does not reach any stub where the register is live, then it is truly dead and we can eliminate it without any further work.

For all the stubs that the definition may reach, we check if the inputs or output of the dead candidate are possibly redefined on any path from the dead candidate to the stub. If this happens, the dead candidate cannot be removed. If not, then we eliminate the dead candidate in the hot routine and place a copy on every stub that the definition reaches.

With register reallocation, we could weaken the condition that requires that the inputs and output of the dead candidate not be overwritten in the path from the operation to the stub. To gauge how much better we could be doing, we estimated the path length reduction possible by assuming that all dead candidates could be eliminated. The results are in Figure 7. The bottom bar is what is actually achieved, and the top bar represents the results of this optimistic assumption. Our results slightly underestimate the gain, because we do not consider the effect of this optimization enabling other optimizations. The potential gain appears to be fairly small and we decided it was not worthwhile trying to remove these extra instructions. From looking at programs, the dead operations that cannot be eliminated are often copies of the argument registers into preserved registers. Interprocedural register allocation or inlining is probably the best way to attack this problem.

5.3 Lifetime Splitting

Often, a lifetime will start in the frequent part of the code and cross into the infrequent part. If the lifetime crosses a call, then it is typically assigned a preserved register, which requires a save and restore. If the call is only in the infrequent part, then it would be better to keep the value in a scratch register in the frequent part, then transfer it to a preserved register in the infrequent part. Then, the save and restore are not executed if the code stays in the frequent part.

5.4 Copy propagation

When we only copy the hot part of a routine, we eliminate some flow paths, making copy propagation legal where it was not before. For example, an argument register may be copied to a preserved register so that its value will be live across a call (e.g. mov a0,s0). However, if the call is eliminated because it is in the infrequent path, then applying copy propagation replaces uses of the preserved register with the original argument registers (e.g. addl s0,1,t0 becomes addl a0,1,t0). Copy propagation often eliminates all uses of a preserved register, which leads to further optimization.

5.5 Eliminating unnecessary preserved registers

Often HCO eliminates all uses of a preserved register by partitioning or by the optimizations described above. If the save and restore is made unnecessary, then it is eliminated in the hot routine and a copy of the save is placed on every stub.

5.6 Eliminating unnecessary stack adjusts

If all references to the stack pointer are removed from a routine, then the stack adjusts on entrance and exit can be removed. For compensation code, a stack adjust to grow the stack is placed on every stub.

5.7 Phase ordering

The order that the optimizations are applied is important for the final result. A sequence of copy propagation, dead code removal, and peephole optimizations is run repeatedly until no further progress is made. Each of these optimizations create opportunities for the others and themselves, and we have found that one to three passes is usually sufficient. Note that profile information allows us to apply optimization to less than 5% of the program text, so optimizer time is not large.

Next, lifetime splitting is applied once. Lifetime splitting uses an extra scratch register and introduces a copy on the stub, so it is better to eliminate the use of a preserved register with one of the previous optimizations than to split the lifetime.

Running the previous optimizations often eliminates all uses of a preserved register, and now the unnecessary saves and restores of preserved registers are eliminated. Finally, if all references to the stack pointer have been removed, we remove the stack adjusts.

6. Related work

Dead code elimination in HCO is closely related to partial dead code elimination[Knoop94], which uses code motions to move an instruction later to avoid computing it. When moving an instruction, partial dead code elimination finds the optimal points in the flow graph to minimize execution, while we take the simpler approach of using profile information to decide if it is profitable to move the instruction to a stub. There is a limited form of partial dead code elimination for superblocks[Chang91], but we have found that the scope of a superblock is too small for effective optimization, even when using techniques that expand their scope such as predication[Mahlke92].

Partial dead code elimination does not alter the flow graph when placing operations. This is done by HCO when it copies the hot part of a routine, which allows us to do better than the “optimal” method in some cases. For example, in Figure 8, the value of b is computed outside of a loop and is only used inside of the conditional inside of the loop. Techniques that don’t alter the program structure can either leave the computation of b outside of the loop where it is always computed once, or put it inside of the conditional, where it is computed once for every time that p[i] is true. If the condition p[i] is rarely true, then HCO moves the computation of b to the stub. If p[i] is never true, b is never computed. The first time p[i] is true, we jump to the stub, compute b and then jump into the middle of the loop, so b is never computed again.

Original routine

```
b = e + c * d
for (i = 0; i < n; i++)
      if (p[i])
         output(b);
return;
```

After HCO

```
for (i = 0; i < n; i++)
      if (p[i])
         goto stub;
return;
stub:
b = e + c * d;
goto inloop;

b = e + c * d;
for (i = 0; i < n; i++)
      if (p[i])
inloop:      output(b);
return;
```

Figure 8: Altering program structure gives HCO more flexibility

Much work has gone into profile directed interprocedural register allocation [Wall86, Santhanam90, Kurlander96]. We consider HCO to be a framework for optimization, and a register allocator for this framework is being implemented. Optimizations such as shrink wrapping [Chow88] and ORA [Goodwin96] have sought to reduce unnecessary spills and reloads by placing them in the program where they are executed less often. Many possible placements are not permitted by the Alpha NT run-time model[+]. HCO considers a smaller set of permissible locations, the stubs. Like partial dead code elimination, these techniques do not alter the flow graph, and the ability to alter the flow graph can enable HCO to find opportunities that other methods cannot.

Region-based compilation[Hank95] uses profile information and aggressive inlining to expose more optimization candidates to the compiler. It then repartitions a program into regions to make compilation of the much larger program tractable. Our approach is similar in that we partition each routine into a hot and cold region. However, the subject of our work is optimizations that push work out of the hot region into the cold region and transformations that make it possible. If we were to add aggressive partial inlining to NT OM, we could use Hank’s methods to partition the program into multiple hot regions.

7. Conclusions and future work

Many commonly used Alpha NT applications spend most of their time in non-leaf routines and in loops that span multiple procedures. For these applications, the most frequently executed basic block in a procedure is often the first basic block. Knowing this, it is not surprising that about 10% of the time is spent in procedure prologs and epilogs, which is essentially procedure call overhead.

The optimization in this paper is effective in eliminating path length from these types of programs. It does this by optimizing the frequently executed parts of a procedure at the expense of the less frequently executed parts. Much of the gain is dependent upon optimizations that eliminate partially dead code or change the register

[+] The Alpha NT software run-time model defines where in a routine the stack adjusts and the saving of preserved registers can be performed; this enables the exception handler to unwind the stack efficiently[CALLSTD].

assignment, splitting lifetimes and changing preserved registers into scratch registers.

HCO is an attractive framework for profile-directed optimization. In the future, we expect to add partial inlining, interprocedural register allocation, speculative scheduling, prefetching, and other optimizations. HCO also provides a practical solution for applying advanced compiler technology to large application programs. We can look beyond the SPECmarks and address the problems in real applications that are in day-to-day use.

Acknowledgments

The authors would like to thank David Goodwin for implementing the interprocedural dataflow in NT OM and to Mark Davis, Joel Emer, David Goodwin, Tryggve Fossum and the reviewers for their comments on this paper.

References

[Blickstein92] D. Blickstein, et al, "The GEM optimizing compiler system," *Digital Technical Journal*, 4(4):121-136.

[Calder94] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," *in ASPLOS VI Proc.*, San Jose, CA, Nov. 1994

[Calder95] B. Calder, D. Grunwald, and A. Srivastava, "The predictability of branches in libraries," *in Proc. of the 28th Annual Intl. Symp. on Microarchitecture*, pp. 24-34, Ann Arbor, MI, Nov. 1995

[CALLSTD] Alpha NT Calling Standard. http://www.partner.digital.com/www-swdev/pages/Home/TECH/documents/alpha_cookbook/biblio.htm

[Chang91] P.P. Chang, S.A. Mahlke, and W.W. Hwu, "Using profile information to assist classic code optimizations," *Software Prac. and Exp.*, 21(12): 1301-1321, 1991

[Chow88] F.C. Chow, "Minimizing register usage penalty at procedure calls," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '88, ACM pp.* 85-94, Atlanta, GA, June 1988

[Dragon] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[Goodwin96] D. Goodwin and K. Wilken, "Optimal and near-optimal global register allocation using 0-1 integer programming," *Software-Practice & Exp.*, To appear 1996.

[Hank95] R.E. Hank, W.W. Hwu, and B.R. Rau, "Region-based compilation," *in Proc. of the 28th Annual Intl. Symp. on Microarchitecture*, pp. 158-168, Ann Arbor, MI, Nov. 1995

[Hwu89] W.W. Hwu and P.P. Chang, "Achieving high instruction cache performance with an optimizing compiler," *in Proc. 16th Annual Intl. Symp. on Computer Architecture*, Jerusalem, Israel, June 1989

[Knoop94] J. Knoop, O. Rüthing, B Steffen, "Partial dead code elimination," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '94, pp.* 147-158, Orlando, FL, June 1994

[Kurlander96] S.M. Kurlander and C.N. Fischer, "Minimum cost interprocedural register allocation," *in The 23rd ACM SIGPLAN SIGACT Symp. on Principles of Programming Languages*, pp. 230-241, St. Petersburg, Florida, Jan., 1996.

[Larus95] J.R. Larus and E. Schnarr, "EEL: Machine-independent executable editing," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Impl. '95, pp.* 291-300, La Jolla, CA, June 1995

[Mahlke92] S.A. Mahlke, et al., "Effective compiler support for predicated execution using the hyperblock," *in Proc. of the 25th Annual Intl. Symp. on Microarchitecture*, pp. 45-54, Dec. 1992.

[McFarling89] S. McFarling, "Program optimization for instruction caches," *in ASPLOS III Proc.*, pp. 183-193, Boston, MA, April 1989.

[Pettis90] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning" *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '90*, pp. 16-27, White Plains, NY, June 1990

[Santhanam90] V. Santhanam and D. Odnert, "Register allocation across procedure and module boundaries" *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '90*, White Plains, NY, June 1990

[Sites95] R. L. Sites and S. Perl, "PatchWrx -- A dynamic execution tracing tool," http://www.research.digital.com /SRC/personal/Dick_Sites/patchwrx/PatchWrx.html.

[Srivastava94] A. Srivastava and D. Wall, "Link-time optimization of address calculation on a 64-bit architecture," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '94*, pp. 49-60, Orlando, FL, June 1994

[Srivastava94b] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools," in Proc. SIGPLAN 94 Conf. on Programming Language Design and Implementation, pp. 85-96, Orlando, Florida, June 1994

[Wall86] D.W. Wall, "Global register allocation at link time" *in Proc. SIGPLAN 86 Symp. on Compiler Construction*, pp. 264-275, Palo Alto, CA, June 1986

[Wilson96] L.S. Wilson, C.A. Neth, M.J. Rickabaugh, "Delivering binary object modification tools for program analysis and optimization," volume 8,1 of *Digital Technical Journal*, pp. 18-31, 1996