Appears in the Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO-31), pages 115–123, December 1998.

Better Global Scheduling Using Path Profiles

Cliff Young
cyoung@research.bell-labs.com
Bell Labs
Lucent Technologies
Murray Hill, NJ 07970

Michael D. Smith
smith@eecs.harvard.edu
Div. of Eng. and Applied Sciences
Harvard University
Cambridge, MA 02138

Abstract

Path profiles record the frequencies of execution paths through a program. Until now, the best global instruction schedulers have relied upon profile-gathered frequencies of conditional branch directions to select sequences of basic blocks that only approximate the frequently-executed program paths. The identified sequences are then enlarged using the profile data to improve the scope of scheduling. Finally, the enlarged regions are compacted so that they complete in a small number of cycles. Path profiles remove the need to approximate the frequently-executed paths that are so important to the success of the compaction phase. In this paper, we describe how one can modify a trace-based instruction scheduler, and in particular a superblock scheduler, to use path profiles in both the selection and enlargement phases of global scheduling. As our experimental results demonstrate, the use of more detailed profile data allows the scheduler to construct superblocks that are more likely to avoid early exits. This effect leads to more useful speculative code motions and an overall improvement in program performance. We also describe how a path-profile-based approach can simplify the engineering of a trace-based scheduler by unifying several trace-enlargement heuristics into a single general mechanism.

***Keywords**: path profiling, global instruction scheduling, superblock scheduling*

1 Introduction

Global instruction scheduling groups and orders the instructions of a program so that the sequence in which instructions are fetched matches the hardware resource constraints while still maintaining the program semantics. Trace-based scheduling [4,8,10] is a common approach to global instruction scheduling that relies on the identification of a sequence of program basic blocks, a *trace*[1], that are frequently executed together. All existing approaches to trace-based scheduling use profiles of control-flow-graph (CFG) edge frequencies to construct approximations of the actual paths encountered during program execution. Recently, several researchers [1,2,22] have proposed efficient techniques for collecting path profiles—profiles that associate execution frequencies with program paths. In this paper, we describe how to modify a trace-based scheduler to use path profiles. In particular, we adapt the well-known superblock scheduling algorithm [8] to use general path profiles [22], and we discuss the engineering effects and quantify the run-time performance benefit of such a change.

A trace-based scheduler consists of two major phases: trace *formation* and trace *compaction*. In the formation phase, traces are first *selected* by identifying frequently-executed program paths. Then these traces are *enlarged* by making extra copies of likely successor blocks. The goal of trace formation is to find program blocks that are frequently executed together, so that work in the later blocks can be usefully overlapped with work from earlier blocks. In trace compaction, the instructions comprising each trace are reordered into a tight execution schedule that matches the hardware resource constraints while maintaining program semantics.

While the engineering of the compaction phase of a trace-based scheduler is complex, the success of this phase depends critically upon the forming of "good" traces. A trace is good if it contains a large number of instructions and if the dynamic program flow often reaches the end of the trace. As discussed in many previous papers on trace-based scheduling, traces with a large number of instructions lead to better schedules because, in general, they provide the compactor with a potentially larger pool of independent instructions. Furthermore, a trace that is likely to complete is preferable to one where the program execution exits before the end of the trace, since the most aggressive compaction algorithms aim to minimize the cycle count for the entire trace. Early exits lower performance: instructions moved before the early exit are wasted work. Our work shows that the use of path profiles can improve the selector's ability to identify and form good traces.

Existing selectors identify high-frequency program paths by knitting together independent edge frequencies. Edge profiles are an example of *point profiles*—profiles that independently aggregate information about particular program points. Using path profiles, we can determine not just the frequency of the block at the start of a trace but also the exact frequency with which all blocks in the trace will be executed. With this information, we can better determine which traces are worthwhile to enlarge, because execution frequently reaches their last block, and which traces will not benefit from enlargement, because execution frequently

1. For clarity, we use the term "trace" when describing a sequence of program blocks that comprise a scheduling region. We use the term "path" when discussing a sequence of blocks followed during program execution.

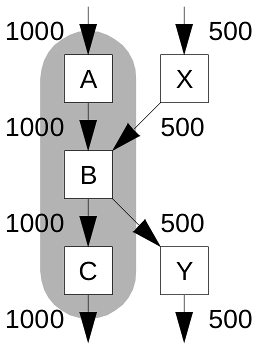

Figure 1. Example of inaccuracies in trace counts under edge profiles. ABC is the selected trace, while XB is a side entrance edge and BY is a side exit edge. Let f(t) be the frequency with which trace t was executed. From the edge profiles, we know that 500≤f(ABC)≤1000, while 0≤f(ABY)≤500. We also know that f(ABC)+f(ABY)=1000 because ABC and ABY are the only paths originating at A. If f(ABC)=1000 (so f(ABY)=0), then ABC is certain to complete, while if f(ABC)=500 (so f(ABY)=500), then ABC is only 50% likely to complete.

takes an early exit. Figure 1 illustrates the difficulty of using edge profiles to approximate this same information. Difficulties arise when frequently executed paths intersect; the intersection results in multiple paths contributing to the aggregated edge frequencies. With edge profiles, a selector is forced to summarize a trace's completion frequency as a frequency range, rather than as a single numerical frequency estimate. If a trace has enough side entrances or exits of large frequency, an estimate based on edge profiles quickly becomes useless.

We begin in Section 2 with a detailed description of trace formation using point and path profiles. To better ground the discussion, we present this in the context of superblock scheduling. The same general heuristical techniques used to select and enlarge superblocks can be adapted in a straightforward manner for use in other kinds of trace-based schedulers. Section 3 describes our experimental methodology, including an brief overview of our algorithm for efficient path-profile collection. In Section 4, we present and discuss our experimental results that illustrate and explain the runtime performance benefit of path-profile-driven instruction scheduling. Section 5 reviews the related work in this area, while Section 6 summarizes the conclusions of this work and considers the ramifications of this work on hardware techniques such as trace caches [13,16].

2 Superblock formation

A superblock scheduler [8] is a type of trace-based scheduler that builds a special kind of scheduling region called a *superblock*. A superblock is a sequence of basic blocks that has only a single entry point (through the top of the first basic block) but may have multiple exit points. Superblocks simplify the bookkeeping required to maintain program semantics during a global code motion.

Superblock scheduling consists of two major phases: formation and compaction. This section concentrates on the techniques for forming superblocks that can be profitably compacted. We first describe how point profiles have been used to form superblocks, then we show how path profiles can be used to form better superblocks. We conclude this section with a summary of our overall implementation of superblock scheduling. A detailed discussion of superblock compaction is well-documented elsewhere [8].

2.1 Formation using point profiles

Superblock *formation* consists of three steps. First, a *trace-selection* step partitions the blocks of a procedure into traces; these traces may have side entrances. Second, a *tail-duplication* step transforms each trace into a superblock by performing code duplication to remove any side entrances. Third, a *superblock-enlargement* step appends copies of a superblock's most-likely successor blocks to the superblock if heuristics indicate that it would benefit from enlargement. Our discussion focuses on the first and third steps; the selection and enlargement steps concern themselves with finding useful blocks to place in a superblock. Tail duplication [8] serves only to transform traces into valid superblocks (considerably simplifying the later work of the compactor).

The partitioning of blocks into traces during trace selection is typically done using a heuristic that identifies blocks as *mutual-most-likely* [10]. For two blocks **A** and **B** to be mutually-most-likely, an edge profile must show that **B** is the most-likely immediate successor of **A** and the **A** is the most-likely immediate predecessor of **B**. A trace is the maximal sequence of mutual-most-likely blocks, with the caveat that no trace can contain a back edge. The designers of the MultiFlow compiler experimented extensively with other trace selection heuristics but found that mutual-most-likely gave the best performance [10].

After trace selection and tail duplication, the resulting superblocks may contain fewer instructions than desirable. This is caused by two factors: traces cannot contain back edges, and the partitioning of blocks into initial traces means that a block cannot belong to more than one trace. To mitigate these problems, the superblock enlarging step extends selected superblocks with copies of their successor blocks. It is important to extend only those superblocks that will benefit from enlargement. Enlarging superblocks by code duplication increases the size of the program and thus can hurt memory system performance. Furthermore, enlarging a superblock that rarely completes (i.e. frequently takes an early exit during program execution) will not help to reduce the program's cycle count, and it can even negatively impact performance if the enlargement process results in a compacted superblock with a longer cycle count to the early exits.

The IMPACT designers list three distinct superblock-enlarging optimizations: branch target expansion, loop peeling, and loop unrolling [8]. In *branch target expansion,* the last branch in each superblock is examined. If the branch is likely to jump to the start of another superblock, the contents of the target superblock are appended to the first superblock. Loop peeling and loop unrolling apply to *superblock loops*, which are superblocks whose last blocks are likely to jump to their first blocks. *Loop peeling* addresses superblocks that iterate a small number of times. To peel a loop, copies of the loop body are made and connected to form a prologue to the original superblock loop. *Loop unrolling* addresses superblocks that iterate a large number of times. Once again, copies of the loop body are made, but the branch of the last copy is connected to the first copy, creating a much larger loop.

Superblock loop peeling and unrolling both assume that each loop in the program has a single dominant path that accounts for most of the iterations through the loop. Whenever a different path through the loop executes, the program will exit the scheduled superblock early. Secondary or tertiary paths through loops will lead to such early exits. As shown in Section 4, path profiles allow us to identify and exploit secondary paths and alternating behavior, while still performing well on loops with single dominant paths.

2.2 Formation using path profiles

With point profiles, the mutual-most-likely heuristic and superblock-enlargement optimizations attempt to construct traces that correspond to high-frequency paths. Since path profiles provide exact path frequencies, no heuristics need to be used to find high-frequency paths. After a brief overview of the specifics of our path profile data, this section describes how we have adapted the superblock formation steps discussed in Section 2.1 to use path profile data.

Informally, a path is a bounded-length sequence of program points; a path profile therefore records execution frequencies for each sequence of invoked procedures, sequence of basic blocks, or sequence of instructions through the program. For this paper, we generate path profiles consisting of basic-block sequences. While a point profile records independent statistics about program points, a path profile captures the dynamic relationships between program points. Though these dynamic relationships make path profiles larger than point profiles, we can use this additional information to perform more beneficial optimizations. Path profiles provide a superset of the information captured by point profiles: one can derive any desired point statistic by summing the frequencies of all paths starting (or ending) at the desired point.

Prior work in path profiling collected execution frequencies for *forward* paths through a CFG [1, 2]. The restriction on a forward path is that it cannot contain a back edge. As we explain in a moment, we collect execution frequencies for *general* paths [22] in this work. A general path can contain any contiguous sequence of CFG edges up to a limiting path length; we use a path length of 15 branches, i.e. a path can contain up to 15 conditional or multiway branches. Profiling for forward paths chops an execution trace of CFG edges into pieces broken at back edges, while profiling for general paths observes a sliding window of branches in the CFG-edge trace. Because back edges end a forward path, a block can appear at most once in a forward path, while a single block may appear multiple times in a general path.

We can use either kind of path to select traces that will become superblocks, since the initially selected traces do not include back edges. However, by using general paths instead of forward paths, we can better understand how to perform superblock enlargement. General paths can include back edges, so general path frequencies remain exact for traces that cover more than a single iteration of a loop. And general paths can capture branch correlation that spans multiple loop iterations, while forward paths cannot. For the rest of this paper, when we refer to a path profile, we mean a general path profile. We use the path profile information both to select initial traces and to enlarge the resulting superblocks.

Up to the profiling depth, path profiles provide exact execution frequencies for each potential trace. Rather than choosing mutually-most-likely predecessors or successors when growing a trace, our path-based trace selector determines exactly how much execution frequency would be lost by extending the trace by a particular node. It begins by tiling the CFG with traces in a manner similar to the mutual-most-likely trace picker. Seeds are selected by node frequency order, as in the edge-profile method. Traces are then grown downward[2] using the most-likely path successor node. The most-likely path successor node is the node that extends the current trace so that the resulting trace has a higher path frequency than the traces produced by the other (if any) successor nodes. For example, suppose that the current trace is **ABC**, that **X** and **Y** are **C**'s CFG successors, and that the frequency of path **ABCX** is 100 while the frequency of path **ABCY** is 200. Then **Y** is **ABC**'s most-likely path successor node. As with mutual-most-likely, traces are terminated when the most-likely successor is already in some trace or the most-likely successor is reached by a back edge. This produces a partitioning of a procedure's blocks into traces. Figure 2 summarizes our algorithm for path-based trace selection.

Our path-based superblock enlarger differs from the enlargement techniques mentioned in Section 2.1 in two major ways. The first difference is in how we select superblocks to enlarge. Since path profiles tell us the exact frequency with which each initial superblock completes, we enlarge only those superblocks that complete with (a user-specified) high frequency. The second difference is in how we go about enlarging a superblock. We use a single enlargement strategy that unifies all three of the previously mentioned optimizations and is furthermore able to exploit branch correlation [14,20]. We extend the end of a superblock by choosing succeeding blocks to duplicate based on the same most-likely-path-successor criterion that we used during trace selection. If our path profiles are not long enough to give exact frequencies for the entire superblock,

2. We have not included support for upward trace growth in our current implementation. It is possible to extend our path-based selector with this capability, though analysis of the output of our current implementation leads us to predict that this additional capability will not noticeably improve the performance of our scheduled code.

```
Block most_likely_path_successor(Trace t) {
  let b_last = the last block in t;
  let b_next = nil, max_f = 0;
  foreach s ∊ successors(b_last) {
    if (f(t·s) > max_f) {
      b_next = s;
      max_f = f(t·s);
    }
  }
  return b_next;
}

Trace select_trace(Block seed) {
  Trace t = seed·nil;
  while (true) {
    Block s = most_likely_path_successor(t);
    if s does not belong to any other trace
      and s is not reached by a back edge;
      t = t·s;
    else break;
  }
  return t;
}

Trace enlarge_trace(Trace t) {
  int c = 0;// loop unrolling count
  while (true) {
    Block s = most-likely-path-successor(t);
    if s is a superblock head block {
      if s is not a superblock loop head block
        or (c++ >= 4)
        return t;
    }
    t = t·s;
  }
}
```

Figure 2. Algorithms for path-based superblock formation and enlargement. The dot operator concatenates two traces to create a single larger trace.

then we use the longest suffix of the superblock for which we have exact frequencies to choose the next block. Notice that this strategy captures correlation: if a superblock ends in a correlated branch and the superblock includes the predictive path for this correlated branch, then we will extend the superblock to the most-likely correlated successor.

Each time we find a most-likely path successor that is also a superblock head, we consider whether to stop enlarging the candidate. As currently implemented, we stop at any head of a superblock that is not a superblock loop, when we encounter a fifth superblock loop head, or when the superblock's instruction count exceeds a preset threshold. Figure 2 also summarizes our path-based enlargement algorithm.

By allowing path enlargement to include four superblock loops, we allow path-based enlargement to perform a transformation similar to unrolling or peeling. Our approach however is more powerful than classical superblock unrolling and peeling; Figure 3 illustrates two motivating examples for loop unrolling.

Path-based enlargement performs branch target expansion for candidates that are not superblock loops. For superblock loops, correlation and history depth conspire to produce loop peeling or loop unrolling. In the case of low iteration count loops, path history will include the most common number of iterations of the loop and will cause the path-driven superblock enlarger to predict the loop to exit after peeling that many iterations. In contrast, high iteration

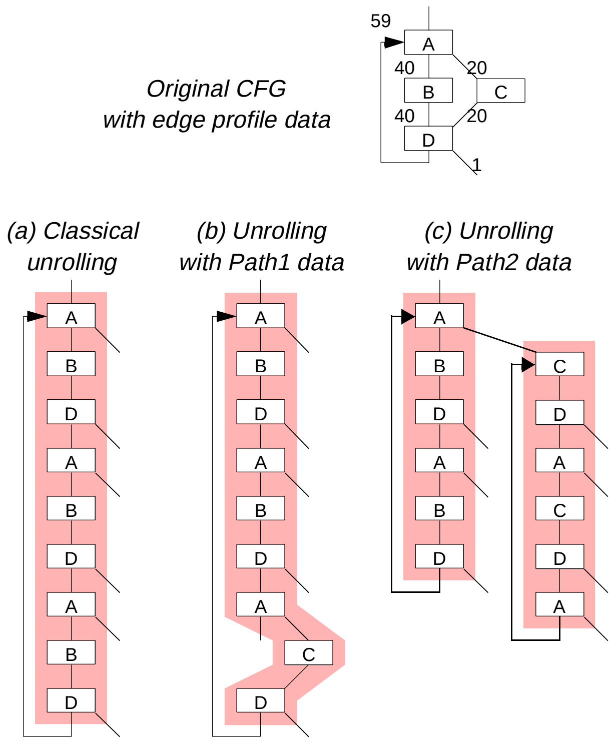

*Figure 3. Classical compared to path-based superblock unrolling. The original profiled loop contains a conditional statement. Unrolling (a) shows the classical result of unrolling this loop three times based on the observed edge profile; the shaded region is the resulting superblock. Given edge profiles, the only sensible unrolling is to create loop bodies containing block **B**. Unrollings (b) and (c) show two different enlargements of the same loop based on two different path profiles. In the Path1 profile, we find that the loop repeats the trace **ABDABDACD** 20 times. In the Path2 profile, the loop's behavior is phased—execution goes left 40 times and then goes right 20 times. Note that both of these path profiles produce the recorded edge profile.*

count loops will appear to stay in the loop on all paths within the history depth. In such cases the path-based loop enlarger will create a large unrolled loop.

By itself, path-based superblock enlargement can be seen as a simplifying engineering technique: where three superblock enlarging optimizations had to be implemented before, now only one is needed. We compare results from selecting regions traditionally and with our path-based approach in Section 4.

2.3 Our superblock implementation

We implemented our instruction scheduler as two compiler passes: a superblock selection/enlargement (*form*) pass and a superblock compaction pass (*compact*). The *form* pass supports basic-block, edge-profiled, or path-profiled region selection. It also optionally provides tail duplication, superblock loop unrolling, superblock loop peeling, superblock branch target expansion, and path-driven superblock enlargement. Our experimental results use the same *com-*

pact pass for both edge- and path-profile-based superblock scheduling. We apply similar thresholds to both scheduling approaches. Exact details are provided by Young [22]. The compaction pass, *compact*, performs top-down cycle scheduling on each superblock.

Compact implements three forms of register renaming to remove dependences and improve the resulting schedules. *Anti and output dependence renaming* work in the standard way to remove such dependences within a superblock. *Live off-trace renaming* performs similar renaming for registers whose values are used off-trace. This allows more instructions to be above superblock exits. *Move renaming* substitutes a source of an instruction that depends upon the result of an unscheduled move with the source of that move so the instruction can be scheduled earlier than the move. We currently support only a limited load and store reordering. Experiments with pointer analysis information [19] improved the code produced by all scheduling approaches but did not change the conclusions of the paper.

The high-level flow of the back-end compilation process is as follows. First, we form superblocks using the edge or path profile-driven formation method described above. For each superblock, we then perform value numbering and dead-code elimination, preschedule using an infinite register variant of the target architecture, allocate registers, postschedule restricted by the allocation decisions, and finally perform a Pettis and Hansen-style [15] procedure-placement optimization. Young [22] provides the full details of our compilation process.

3 Methodology

This section describes our evaluation system and the benchmarks we used in our experiments. We begin with a brief description of our path profiler, which uses an algorithm that differs from those published previously. We then describe our experimental machine model and the compiled simulation techniques we use to measure the performance. Lastly, we describe the benchmarks on which we measured performance.

3.1 Efficient general path profiling

To collect profiles, our compiler [17,21] inserts instrumentation code that observes each executed CFG edge in the program, then passes information to an analysis routine that is linked with the program. By linking different analysis routines, we can change the analysis performed at run-time. We implemented both an edge profiler and an efficient path profiler.

As mentioned above in Section 2.2, we collect general path profiles rather than forward path profiles. To collect general profiles efficiently, we exploit two observations about general paths. First, the number of successors to a path is small: if we observe path **ABCD** and the CFG successors of **D** are **X** and **Y**, then the only possible next path will be either **BCDX** or **BCDY**. Second, we do not expect to execute all possible paths through a program, so we can lazily explore the space of possible paths. By constructing path descriptions (and frequency statistics) lazily and by keeping pointers to successor paths, we do *O(npaths + nedges)* work over the course of a run, where *npaths* is the number of distinct paths seen during the run and *nedges* is the number of dynamic edges processed by the analysis routines. If the number of paths is much smaller than the number of dynamic edges, this averages out to *O(1)* work per executed edge, which is the same overhead as edge profiling. Young [22] presents the full details of this profiling approach and a more careful analysis of its overhead.

3.2 Experimental machine model

We simulate a very powerful machine VLIW model based on the Digital Alpha ISA. We extend the instruction set with non-excepting instructions. The machine includes 8 functional units that can execute any kind of instruction in a single cycle; however, we limit the machine to only one control instruction per cycle. We assume an integer register file with 128 registers; additional registers are allocated equally to extend the caller- and callee-saved registers of the Digital UNIX calling conventions. We used an instruction cache size of 32KB (direct-mapped with 32 byte lines), and we ignore data cache effects (data memory traffic will increase under scheduling because of speculative loads).

Compared to actual Alpha implementations, our experimental machine is very powerful. In defense of our machine model, trends in processor design are moving toward this idealized machine. Instruction latencies appear to be shortening and issue widths are widening; register files are growing larger even though the number of architected registers remains fixed. We have also generated results with more realistic instruction latencies, and we found that the benefit of path-profile-based scheduling increased.

Since we target a hypothetical machine model, we use compiled simulation to measure the performance of our benchmark applications. In the compiled simulation, we add code that installs an operating system trap handler to suppress exceptions caused by unsafe speculative instructions. The programs generated by our compiler run on a real Digital Alpha and produce correct results.

3.3 Benchmarks

Table 1 lists our benchmark applications with some descriptive information. In addition to SPEC benchmarks, we include a small number of “microbenchmarks” that are idealized examples of the kind of behavior that can be exploited using path profiles but is invisible using point profiles. Most microbenchmarks take no input; they list “null” as the testing input. For all benchmarks with inputs, we use different training and testing data sets, though we do not list the training sets in Table 1.

Table 1 also lists the dynamic branch, cycle, and instruction counts for each testing data set. Branch counts were collected using the branch instrumentation described earlier, while the cycle and instruction counts were collected using our compiled simulator of a basic-block scheduled version of the program running on the experimental machine model

	Benchmark	Description	Testing Data Set	Size (KB)	Dynamic Counts (in Millions)		
					Branches	Cycles	Instr's
micro	alt	Sorted example	null	25	2.0	8.5	12.3
	ph	Phased example	null	25	2.0	9.0	11.3
	corr	Branch corr. example	null	25	0.5	1.7	2.2
	wc	UNIX word count program	PostScript conference paper	25	14.6	52.1	60.2
SPECint92	com(press)	Lempel/Ziv file compression	MPEG movie data (6MB)	57	135.4	827.4	1,263.5
	eqn(tott)	Translates boolean eqns to truth tables	priority encoder, SPEC92 ref input	115	335.8	1,576.5	2,440.3
	esp(resso)	Boolean minimization	tial, SPEC92 ref input	565	157.2	785.4	1,145.4
SPECint95	gcc	GNU C compiler	cccp.i, SPEC95 ref input	5,595	244.1	1,307.3	1,905.4
	go	Plays the game of Go	9stone21, SPEC95 ref input	918	4,177.3	28,259.3	43,234.1
	ijpeg	JPEG encoder	vigo, SPEC95 ref input	573	1,801.3	20,253.6	51,485.9
	li	XLISP interpreter	SPEC95 ref input	279	10,961.5	57,778.4	80,503.0
	m88k(sim)	Microprocessor simulator	dhry, SPEC95 test input	532	116.0	630.8	886.7
	perl	Interpreted programming lang.	primes, SPEC95 ref input	1,032	2,274.6	12,986.6	19,459.3
	vortex	Object-oriented database	SPEC95 test input	1,737	1,068.8	6,456.8	12,568.2

Table 1: Benchmarks, data sets, and some statistics. The alt *benchmark comprises a single loop containing a conditional that follows the following repeated pattern: TTTFTTTF…. The* ph *benchmark also comprises a single loop containing a conditional, but this time the conditional follows the pattern: TTT…TFFF…F. These two benchmarks produce results under path-based enlargement that are similar to PATH1 and PATH2 in Figure 3. The* corr *benchmark corresponds to the simple correlation example used in Young and Smith [20]. "Size" is the number of bytes in the binary executable file of the original version of the program.*

described in Section 3.2. The major point of the statistics in Table 1 is that our benchmarks run for a significant amount of time and are reasonable to use in branch prediction and instruction scheduling studies.

4 Results

We begin by presenting cycle-count results obtained using our experimental machine model with a perfect instruction cache. Figure 4 illustrates that we can achieve a reduction of 2–16% for the SPEC benchmarks when performing superblock formation and enlargement using path profiles instead of edge profiles. We limit both scheduling approaches to an unrolling factor of 4. As expected, the microbenchmarks demonstrate greater reductions than the SPEC benchmarks, since we constructed the microbenchmarks to show the benefit of path-based formation.

These results are encouraging, but to understand better the potential performance benefit of path profiles, we also measured the impact of our scheduling optimizations on the miss rate of a primary instruction cache as described in Section 3.2. Code expansion is a well-known side effect of aggressive optimizations involving code duplication. And, our simulation results do show somewhat higher miss rates under a path-based approach than under an edge-based one, but not in all cases. Young [22] presents the full set of miss rates. For this paper, we highlight just the results for *gcc* and *go*; these benchmarks have non-trivial miss rates that are noticeably larger under our path-based approach. Under the edge-based approach, we measured a miss rate of 2.67% and 2.53% for *gcc* and *go* respectively, while the path-based approach yielded miss rates of 3.92% and 4.67%. Using a miss penalty of 6 cycles, the "P4" columns in Figure 5 show that, even when we include the impact of the memory system penalties on the cycle count, we still obtain a significant reduction (up to 13% for the SPEC benchmarks) in the cycle count. Unfortunately, one benchmark, *go*, did run slower.

Since we already include a procedure-placement algorithm [15] in our compiler, we decided to adjust the enlargement heuristics of "P4" to see if we could reduce the nontrivial increase in a few of the miss rates. In "P4", all superblocks are treated equally: a superblock, whether it is initially a superblock loop or not, is enlarged until it contains at most 4 superblock loops. "P4e" treats superblocks that are not superblock loops differently: the enlargement of these superblocks ends at the first superblock head (i.e. enlargement uses only tail-duplicated code). Using this heuristic, we were able to outperform the edge-based approach on all of the SPEC benchmarks. Other than the "P4e" experiment, we have not performed any other sort of "SPECmanship" to tune the engineering of the enlargement heuristic so that it performs well over a wide range of applications, though opportunities certainly exist. Chen et al. [3] discuss techniques that address the interaction of code-expanding optimizations and instruction cache miss rates in much more depth.

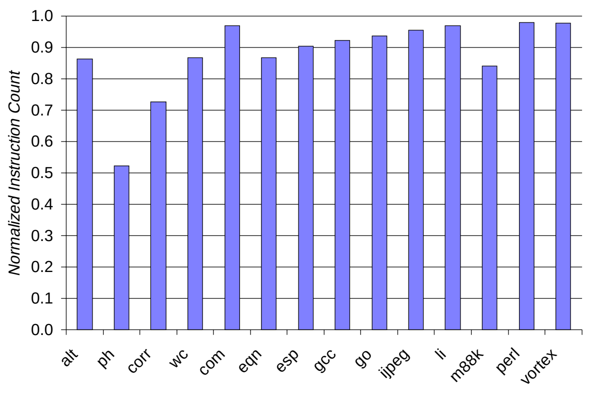

Figure 4. Cycle counts for path-based superblock scheduling normalized against the counts for the edge-based approach. Both approaches are limited to an unroll factor of 4 and both assume an ideal instruction cache.

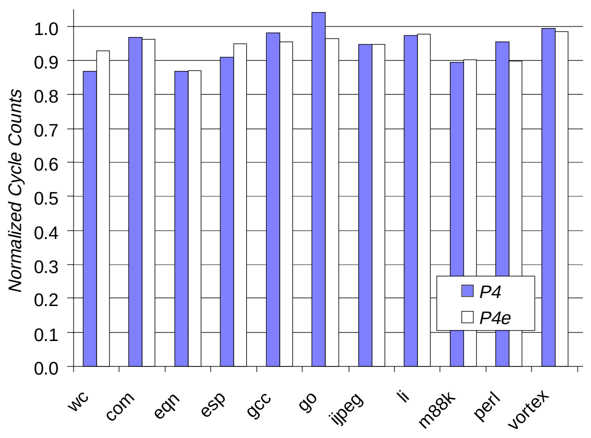

Figure 5. Cycle counts with an 32KB direct-mapped instruction cache. As in Figure 4, the cycle counts for the path-based approaches are normalized against the counts for the edge-based approach. "P4" corresponds to the path-based algorithm described in Section 2.2. "P4e" modifies the enlargement heuristics in order to limit code expansion. All approaches are limited to an unroll factor of 4. We do not display the results for the microbenchmarks since their results are essentially unchanged (they are so small that they always fit in the cache).

We ran one last performance experiment. We were interested in investigating whether it is more important to unroll aggressively or alternatively to exploit the actual important paths (under a more-limited unrolling bound). Figure 6 displays the results of this experiment. The surprising result is that, except for a few benchmarks, exploiting path profile information with an unrolling limit of 4 leads to smaller cycle counts than scheduling using edge profiles and an unrolling limit of 16. Though *eqntott* contains a very high-frequency correlated branch, as first identified by Pan et al. [14], the block guarded by this branch is very small. Hence, loop unrolling is more important to the performance of *eqn-*

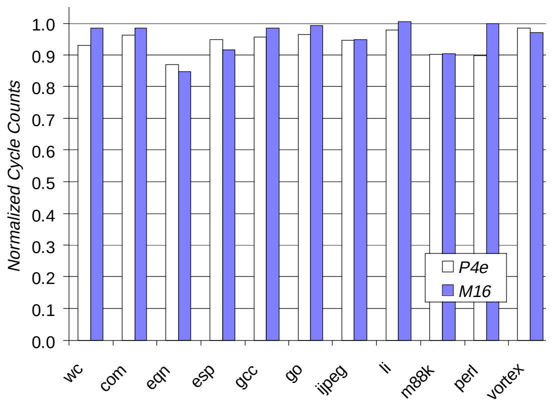

Figure 6. Cycle counts with an 32KB direct-mapped instruction cache. As with the previous graphs, the cycle counts for "P4e" and "M16" are normalized against the counts for the edge-based approach with an unrolling factor of 4. "M16" corresponds to superblock scheduling using edge profiles, the mutual-most-likely heuristic, and an unrolling factor of 16.

tott than the exploiting of branch-correlation for instruction scheduling purposes. The run times of *compress* and *ijpeg* are dominated by few loops, and thus, the dilution of a loop's schedule by the enlargement of a superblock, which is not a superblock loop, into a superblock loop reduces performance. The cycle counts for "M4" and "M16" under *go* and *li* demonstrate that unrolling alone is insufficient when an application's performance is dominated by low iteration count loops and/or frequent procedure calls.

To help explain the cycle-count results, we calculated two other metrics: the average number of basic blocks executed per superblock and the average size of a superblock, both dynamically weighted. Figure 7 summarizes these results. The first of these numbers gives us a feel for how long the execution stays in a superblock before exiting; the first and second together tell us how close we got on average to reaching the end of the superblocks. Given that the compaction phase tries to pack the entire superblock as tightly as possible, we would like the first number ("average") to be as large as possible. To avoid the penalties of unproductive code expansion, we would like the increase the second ("maximum") only if we noticeably increase the first.

The "average" results in Figure 7 support our claims that paths lead to superblock formation where superblocks exit later. Except for *eqntott* where unrolling is important, "P4" produces a bigger "average" run in a superblock than "M16" while often building smaller-sized superblocks. For *go* and *li* whose cycle-count results show little benefit from greater degrees of unrolling, the "average" number of blocks executed per superblock hardly changes between "M4" and "M16". The large "maximum" numbers for the path-based approaches, relative to the edge-based approaches, for *gcc* and *go* help to explain our miss rate findings.

5 Related work

As mentioned above, efficient algorithms to collect for-

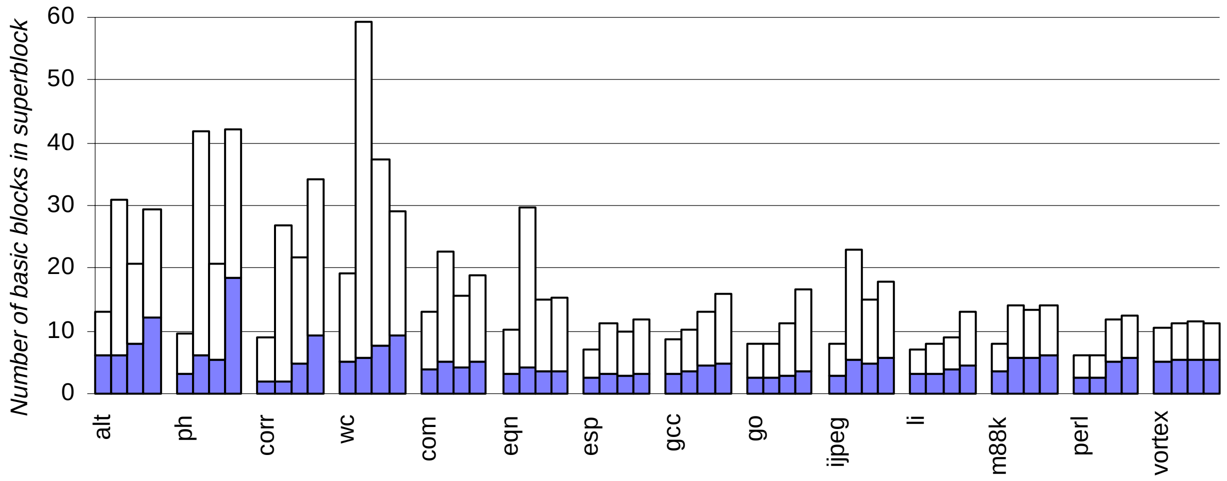

Figure 7. Number of basic blocks executed per dynamic superblock (gray bars) compared to the size in blocks of a dynamic superblock (white extensions). Each benchmark has four bars associated with it; these correspond to the "M4", "M16", "P4e", and "P4" schemes, in that order from left to right. "M4" was the baseline in Figures 4–6.

ward path profiles have been developed by Ball and Larus [2] and by Bala [1]. We know of several other recent compiler optimization efforts that use path profile information: Young and Smith's static correlated branch prediction [20], and Mowry and Luk's correlated data cache miss optimization [11], and Gupta et al.'s work on path-profile-guided partial redundancy elimination [7] and partial dead code elimination [6] techniques. Like our work, each of these techniques uses path information to better evaluate the potential benefit of a code transformation. Gloy et al. [5] use temporal summary information of procedure interleavings to improve instruction cache performance; their profiles, like path profiles, go beyond the aggregation of simple point statistics.

Path-related techniques are not restricted to the software domain. Path information has been used in dynamic branch prediction [12] and dynamic trace prediction [9]. Recently, architecture researchers have proposed the capture of instruction traces in special hardware buffers, called trace caches [16] or DIF caches [13]. The main motivation for these buffers was to improve the fetch efficiency [18] (i.e. to remove the fetch problems caused by taken branches) of the blocks within a dynamic trace. These buffers perform an action similar to the trace-selection step of our trace-formation phase. There is on-going research in the architecture field trying to identify heuristics for effectively identifying and enlarging these dynamic traces so that overlap between trace buffer entries is minimized and so that fetches hit more often in the trace cache.

6 Conclusions and observations

We have shown how to use path profiles in a trace-based scheduler to improve the code produced during global instruction scheduling. Path-based superblock scheduling can achieve lower cycle counts when compared to mutual-most-likely superblock scheduling. Some engineering remains to find the best path-based enlargement heuristic that appropriately weighs the scheduling benefit of code expansion against the instruction cache penalties of this same code expansion. The scheduling benefits result from a path-based superblock scheduler's formation (i.e. selection and enlargement) of superblocks that are more likely to execute to completion. Furthermore, path-based superblock enlargement unifies the separate optimizations of branch target expansion, loop peeling, and loop unrolling into a single optimization; this simplifies the engineering of the superblock enlarger.

7 Acknowledgments

This research was funded in part by a NSF Young Investigator award (grant no. CCR-9457779), a DARPA grant no. NDA904-97-C-0225, and research gifts from AMD, Digital Equipment, HP, IBM, Intel, and Microsoft. Cliff Young was supported by an IBM Cooperative Fellowship.

8 References

[1] V. Bala, "Low Overhead Path Profiling," HP Laboratories Technical Report HPL-96-87, June 1996.

[2] T. Ball and J. Larus, "Efficient Path Profiling," *Proc. 29th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 46–57, December 1996.

[3] W.Y. Chen, P.P. Chang, and W. W. Hwu, "The Effect of Code Expanding Optmizations on Instruction Cache Design," *IEEE Transactions on Computers*, 42(9):1047-1057, Sept. 1993.

[4] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[5] N. Gloy, T. Blackwell, M. Smith, and B. Calder, "Procedure Placement Using Temporal Ordering Information," *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 303–313, December 1997.

[6] R. Gupta, D. Berson, and J. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp. 102–115, November 1997.

[7] R. Gupta, D. Berson, and J. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," to appear in *Proc. IEEE Conf. on Computer Languages*, May 1998.

[8] W. Hwu, et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing* 7(1/2):229–248, Kluwer Academic Publishers, May 1993.

[9] Q. Jacobson, E. Rotenberg, and J. Smith, "Path-based Next Trace Prediction," *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 14–23, December 1997.

[10] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing* 7(1/2):51–142, Kluwer Academic Publishers, May 1993.

[11] T. Mowry and C. Luk, "Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling," *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp.314–320, December 1997.

[12] R. Nair, "Dynamic Path-Based Branch Correlation," *Proc. 28th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 15–23, November 1995.

[13] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," *Proc. 24th Annual International Symposium on Computer Architecture*, pp. 13–25, June 1997.

[14] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 76–84, October 1992.

[15] K. Pettis and R. Hansen, "Profile Guided Code Positioning," *Proc. ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp. 16–27, June 1995.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 138–148, December 1997.

[17] M. Smith. "Extending SUIF for Machine-dependent Optimizations," *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp.14-25, January 1996.

[18] M. Smith, M. Johnson, and M. Horowitz. "Limits on Multiple Instruction Issue," *Proc. 3rd Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pages 290–302, April 1989.

[19] R. Wilson and M. Lam, "Efficient Context-sensitive Pointer Analysis for C Programs," *Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[20] C. Young and M. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proc. 6th Annual Intl. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 232–241, October 1994.

[21] C. Young and M. Smith. "Branch Instrumentation in SUIF," *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 139-145, January 1996.

[22] C. Young, "Path-based Compilation," Ph.D. Thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, January 1998.