Static Correlated Branch Prediction

CLIFF YOUNG
Bell Laboratories, a Division of Lucent Technologies
and
MICHAEL D. SMITH
Harvard University

Recent work in history-based branch prediction uses novel hardware structures to capture branch correlation and increase branch prediction accuracy. *Branch correlation* occurs when the outcome of a conditional branch can be accurately predicted by observing the outcomes of previously executed branches in the dynamic instruction stream. In this article, we show how to instrument a program so that it is practical to collect run-time statistics that indicate where branch correlation occurs, and we then show how to use these statistics to transform the program so that its static branch prediction accuracy is improved. The run-time information that we gather is called a *path profile*, and it summarizes how often each executed sequence of program points occurs in the program trace. Our path profiles are more general than those previously proposed. The code transformation that we present is called *static correlated branch prediction* (SCBP). It exhibits better branch prediction accuracy than previously thought possible for static prediction techniques. Furthermore, through the use of an overpruning heuristic, we show that it is possible to determine automatically an appropriate trade-off between code expansion and branch predictability so that our transformation improves the performance of multiple-issue, deeply pipelined microprocessors like those being built today.

Categories and Subject Descriptors: B.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC*; *VLIW architectures*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Branch correlation, branch prediction, path profiling, profile-driven optimization

The research reported in this article was conducted while Cliff Young was a graduate student in the Division of Engineering and Applied Sciences at Harvard University. Cliff Young was supported in part by a National Defense Science and Engineering Graduate Fellowship from the Office of Naval Research and by an IBM Cooperative Fellowship. Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), NSF grant no. CDA-94-01024, DARPA grant no. NDA904-97-C-0225, and research grants from AMD, Compaq, Digital Equipment, HP, IBM, and Intel. Authors' addresses: C. Young, Lucent Technologies, 700 Mountain Avenue, Room 2C-523, Murray Hill, NJ 07974; email: cyoung@research.bell-labs.com; M. Smith, Division of Engineering and Applied Sciences, Harvard University, 29 Oxford Street, Cambridge, MA 02138; email: smith@eecs.harvard.edu.

1. INTRODUCTION

In both the architecture and compiler domains, conditional branch instructions are a barrier to higher levels of performance. Branches change the program counter based on run-time information. Hardware techniques such as pipelining and multiple issue increase the cost of changing the program counter, making it difficult to resolve the target of a branch instruction in time to keep an execution engine employing these techniques full of instructions. Branch prediction, whether dynamic (hardware-based) or static (software-based), makes good guesses about likely branch targets and allows the instruction unit to fetch instructions early. When predictions are accurate, the execution engine can operate at full speed and performance improves.

Accurately predicting the likely target of conditional branches is also critical for the success of compile-time optimizations, optimizations such as branch alignment [Calder and Grunwald 1994; Pettis and Hansen 1990; Young et al. 1997], global instruction scheduling [Bernstein and Rodeh 1991; Fisher 1981; Hwu et al. 1993; Lowney et al. 1993; Moon and Ebcioglu 1992; Smith 1992; Young and Smith 1998], partial dead-code elimination [Gupta et al. 1997], and partial redundancy elimination [Gupta et al. 1998]. Unlike the execution engine which makes a prediction for each dynamic instance of a static branch in an application, the compiler makes but a single static prediction for each static branch. As before, however, when predictions are accurate, these optimizations are beneficial, and performance improves. In this article, we present a compile-time code transformation that improves the static predictability of conditional branches.

Perhaps the most promising branch-related discovery of the last 10 years is that branches exhibit *correlation*: the outcome of a conditional branch is often determined by the branch's historical pattern of outcomes or the historical pattern of outcomes of its neighboring branches. Due perhaps to the logical structure of the program or aspects of realistic data sets, programs behave in patterns that can be observed, recorded, and exploited. The most successful application of branch correlation has been in dynamic branch prediction, where architects have built predictors that remember patterns in the stream of branch executions [McFarling 1993; Pan et al. 1992; Yeh and Patt 1991; 1993; Yeh 1993; Young et al. 1995].

Our work is related to the recent work in dynamic *global* predictors, predictors that identify and exploit repetitive behavior in the trace of all conditional branches executed by the program. If a branch always goes the same way when reached by a particular pattern of prior branches, we can improve the predictability of this branch by recognizing when the pattern of prior branches occurs. For example, Figure 1 shows a code fragment from the SPECint92 benchmark *eqntott*. In this fragment, the condition of the third branch is always false if the conditions of both previous branches were true. Architects have built hardware structures that exploit such global branch patterns [McFarling 1993; Yeh and Patt 1991]; we show that a compiler can exploit them as well.

Figure 1 demonstrates what we call *logical* correlation; we can logically prove that a must equal b if they both equal 2. Several researchers [Bodik et al. 1997; Mueller and Whalley 1995] have designed static analysis techniques that identify some forms of logical branch correlation and reorganize the code to isolate and

```
if (aa == 2)
   aa = 0;
if (bb == 2)
   bb = 0;
if (aa != bb) {
   ...
}
```

aa == 2	bb == 2	aa != bb
TRUE	TRUE	FALSE
TRUE	FALSE	?
FALSE	TRUE	?
FALSE	FALSE	?

Fig. 1. Example of branch correlation from the SPECint92 benchmark `eqntott` and the corresponding truth table. If the first two branches have true conditions, then the third branch will always have a false condition.

eliminate avoidable conditional branches. Hardware approaches, however, exploit more than just logical correlation. They are also able to exploit what we call *statistical* correlation; a pattern of correlation that is a likely but not guaranteed. We provide examples of statistical correlation and discuss these related techniques in later sections.

We present a static technique for capturing both logical and statistical correlation. To identify all kinds of branch correlation during compilation, we use profile information to summarize a program's run-time behavior. Traditional profiles record statistics for individual points in the program, e.g., the execution frequencies of each static function or instruction. Traditional profiles suffice for per-branch prediction schemes, where the majority direction of each branch point is used as the prediction for that branch point.

A branch prediction scheme that exploits branch correlation requires more detailed execution information than that provided by traditional (point) profiles. As mentioned above, run-time global prediction schemes are based on the recorded history of global branch patterns. A global branch pattern is simply a shorthand[1] for the most recent piece of an execution trace, and this piece simply corresponds to a sequence of basic blocks (or path) through the traced program. This observation inspired a new kind of profile: instead of collecting statistics about the behavior at *points* in the program, why not collect the same statistics about the behavior over *paths* in the program? Path profiles cover the spectrum between traditional point profiles and the brute-force approach of collecting and analyzing complete program traces.

With path profiles, there is the potential for an exponential explosion in the amount of data collected. If our example in Figure 1 had more intervening branches between the first and last conditional, we would have needed a much larger truth table to describe all possibilities. Some mechanism must be used to bound the number of paths collected during profiling. Point profiles and full traces can be seen respectively as path profiles of length 0 or of length proportional to the running time of the program.[2] We give details of how to bound the number of paths in

[1]A branch pattern may not identify a unique trace, since two branches can jump to the same code location and thus produce the same global branch pattern.

[2]A path may include an execution of the same static branch multiple times. This definition of a path is more general than the one used by Ball and Larus [1996]. As discussed in Section 2.2, this more general definition provides us with more opportunities for compile-time optimization.

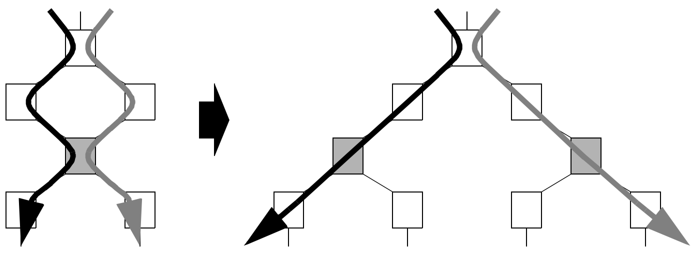


Fig. 2. An intuitive illustration of the SCBP transformation. The left side of the diagram shows a pair of conditional statements and the two most likely paths through them. When reached by the left path, the gray block is likely to go left; when reached by the right path, the gray block is likely to go right. Without code duplication, we can make only one static prediction for the one copy of the gray block. The right side of the diagram shows the subgraph after SCBP. Now there are two copies of the gray block, allowing different static predictions for the different copies.

Section 2. That section also contains a description of an efficient algorithm for collecting path profiles.

In Section 3, we present an algorithm that extracts branch correlation from an application's path profile and then transforms the application so that its correlated branches are more predictable at run time. We refer to this algorithm as *static correlated branch prediction* (SCBP). The transformation is accomplished by duplicating and rearranging blocks of the control-flow graph, making different copies of an original program block represent different correlation history.

SCBP is an instance of the more general idea of specialization. Like other optimizations in this class (e.g., procedure cloning), it makes a space-time trade-off. In particular, SCBP trades memory system performance for improved branch predictability. Figure 2 illustrates a very simple example of this trade-off. In the figure, the second conditional statement is correlated to the first conditional statement: control flow tends to follow either the black or the gray path. With static branch prediction, we must fix a prediction per branch in the program before the program runs. Without duplicating code, the static prediction for the second conditional must favor the left side or the right side, penalizing one of the correlated paths. By duplicating the second conditional statement, we can make different static predictions for the two copies of the statement. With the different copies, we can make more accurate static predictions that exploit correlation, reducing the running time of the program. But the new program is larger than the original program, so it takes up more space. This additional space may hurt performance on a real machine because of the instruction memory resources consumed by the larger program. As we show in Section 3.5, choosing how much space to use to reduce branch mispredictions is key to making SCBP practical.

This work subsumes all prior work on static correlated branch prediction, and it makes several new contributions. Overall, it makes the following contributions:

—We define and explore *general path profiles*, illustrating why they are useful and provide more information than other kinds of path profiles. We appear to have been the first researchers to have collected path profile information [Young and Smith 1994]. Although our initial work focused on improved branch prediction through path profiles rather than on efficient profile collection, path profiling was the key insight that enabled us to improve static branch prediction accuracy.

—We give an efficient algorithm for profiling general paths. This algorithm has the same asymptotic efficiency as current point profiling and tracing techniques. The overhead of our (untuned) implementation is on average within a factor of two of the overhead of point-profiling techniques; tuning of other path-profiling techniques has shown that path profiling can be as efficient as edge profiling [Ammons et al. 1997; Bala 1996; Ball and Larus 1996].

—We describe a practical algorithm, static correlated branch prediction (SCBP), that trades off code expansion for improved branch prediction accuracy. This includes a global reconciliation step, which ensures that the minimum number of copies of each original basic block are made to achieve the desired prediction accuracy.

—We provide a heuristic overpruning method for automatically and effectively tuning the SCBP space-time trade-off to match the cache and branch prediction structure of the target processor.

—We show comprehensive results, including simulation results for SCBP coupled with a code layout pass, and timings of our profiling and optimized programs, running on real machines.

The rest of the article proceeds as follows. Section 2 explains how to collect path profiles efficiently. Section 3 applies path profiles to improve the accuracy of static branch prediction. Section 4 reports on simulated and actual performance of both the profiling and optimization parts of our system. Section 5 discusses related work. Section 6 summarizes our results and concludes.

2. PATHS AND PATH PROFILING

We begin in Section 2.1 by defining some necessary terminology. Section 2.2 gives an example of a simple program and the different kinds of behavior that are captured by various kinds of profiles. The last part, Section 2.3, presents an efficient algorithm for collecting general path profiles.

2.1 Definitions

Programs appear to machines as linear sequences of instructions. Instructions such as conditional branches, jumps, procedure calls, and returns transfer control from one part of the program to another. We refer to these instructions that can change the program counter as *control transfer instructions* (CTIs). For many compiler optimizations, it turns out to be useful to view the program as a graph rather than a linear sequence of instructions, where the graph summarizes all possible control transfers made by the program. We first define the concept of a path in a general graph and then formalize the concept of a path in a control-flow graph. From this discussion, it should be straightforward to apply the concept of a path to other types of graphs used in compilation, e.g., call graphs.

Formally, a graph is a pair (V, E) where V is the set of *nodes* or vertices in the graph, and E is the set of *edges* in the graph. Each edge is an ordered pair of nodes; the ordering is important in directed graphs, where the edges (u, v) and (v, u) are distinguished. The vertices that make up the edge are called the *source* and *target*, respectively; the edge goes from the source to the target. Intuitively, nodes in the graphs that we discuss represent pieces of the program at different granularities, while edges connect nodes that might be executed immediately after one another. For some graphs, it may be natural to extend the definition to include designated start and finish nodes. A *path* is just a finite sequence of edges in the graph, where each edge in the sequence has the same target as the source of the next edge (i.e., the path traces out a sequence of edge-connected vertices in the graph). The number of edges in the path is the length of the path. Note that the number of nodes in the path is one greater than the length of the path.

A *control-flow graph* (CFG) represents the relationships between *basic blocks* [Aho et al. 1988] within a single procedure; a basic block is a maximal single-entry straight-line sequence of instructions. Each CFG edge represents a potential flow of control between the code blocks; control flows from the source to the target of an edge. In a CFG, a block with multiple successors is called a *split* node; a block with multiple predecessors is called a *join* node. CFGs are sometimes built with extra *start* (sometimes called entry) and *finish* (exit) nodes. These indicate where control can enter or exit from the CFG.

In terminological shorthand, we often say that block B *succeeds* block A if there is an edge from A to B in the CFG; one can also define a "successor" relation on CFG blocks that is equivalent to the edge set E. The transitive closure of the successor relation is the "descendant" relation; if B is a descendant of A then there is a path from A to B. The reverse of the descendant relation is the "ancestor" relation. If all paths from the entrance node to node B include node A, then node A is said to *dominate* node B. An edge from L to H where H dominates L is called a *loop* or *back* edge. This corresponds to one intuitive definition of a loop: in order to traverse the edge from L to H, the program must have executed H before.

Some specific kinds of paths are especially relevant to recent work in profiling. A *forward* path is a path that contains no back edges. In contrast, a *general* path (or just a path, but we sometimes refer to general path profilers to distinguish them from forward path profilers) has no constraints on its nodes or edges; it may or may not contain duplicated edges or nodes. A *cycle* is just a path where the source of the first edge is the same as the target of the last edge: it is a path that returns to its start. Cycles in the control-flow graph correspond to loops in the program.

2.2 Example Profiles

We will use a simple example to illustrate the differences between kinds of profiles. Figure 3 shows a C code fragment along with the corresponding part of a control-flow graph.

Point profiles of this code fragment report execution frequencies for CFG nodes or edges. For example, if the initial value of variable `size` is 60, then the node frequencies of A and D will be 60. From the code, we do not know the distribution of values in the data array. For the sake of illustration, suppose that 40 data elements were positive, while the remaining 20 were zero or negative. Figure 4 depicts the point profiles that result, placing frequencies above the node or edge that generated that execution count.

Point profiles indicate the frequently executed basic blocks of the program. From the profiles in Figure 4, it appears that spending effort optimizing the "B" side of

```
int size, data[1000];
...
do {
   if (data[size - 1] > 0) /* A */
      /* action B */;
   else
      /* action C */;
} while (--size); /* D */
```

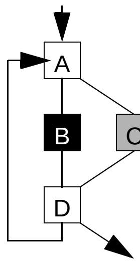


Fig. 3. A code fragment and its corresponding control-flow graph. Each of the nodes in the graph corresponds to a part of the procedure. The node labeled "A" corresponds to the code that evaluates the condition of the *if* statement, while the node labeled "D" holds the code that decrements the variable `size` and tests the decremented value for the *do/while* statement. Nodes "B" and "C" correspond to the *then* and *else* clauses of the conditional statement.

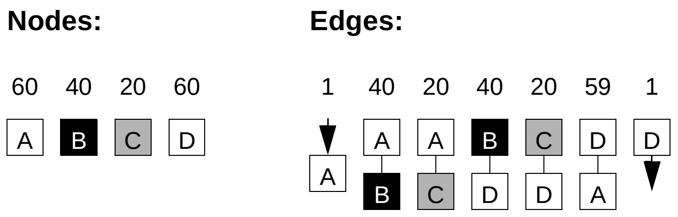


Fig. 4. Examples of node and edge profiles. Each number indicates the execution frequency of the node or edge below it.

the loop will yield greater rewards than optimizing the "C" side of the loop, simply because the "B" side is executed more frequently. Also, optimizations that trade off performance on "C" in favor of improved performance on "B" will improve overall performance, while the reverse is not true.

But point profiles also summarize away much of the run-time complexity of the program. Each counter records *independently*; a point profile records no information about the relations between the times that counters were incremented. From the point profiles, we cannot see the sequence of *if/then/else* decisions across loop iterations. Very different program behaviors can generate the same point profile. For the example in Figure 4, the data might have been sorted in descending order, or the data might have exhibited a repeating pattern of two positive numbers followed by a negative number. These differences in the data would give rise to different program behavior. In the former (*sorted*) case, all 40 "B" iterations will take place first, followed by all 20 "C" iterations. In the latter (*repeating*) case, the iterations will follow a pattern of two "B" iterations followed by a single "C" iteration. Point profiles give no insight into these details of program behavior.

Path profiles capture more detail about program behavior. Figure 5 draws some of the shorter paths that occur in Figure 3 by unwinding paths in the graph into straight lines. Figure 6 shows the subset of the paths that captures the behavior of the conditional statement across two iterations of the loop; these selected paths illustrate how path profiles can capture the differences between the sorted and

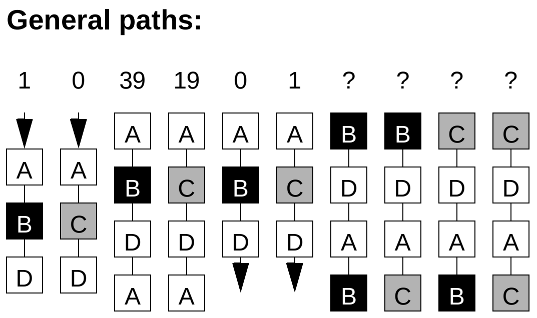


Fig. 5. Example path profiles. The number above each path is its execution frequency during a particular run of the program (frequencies left as question marks will be examined in the next figure).

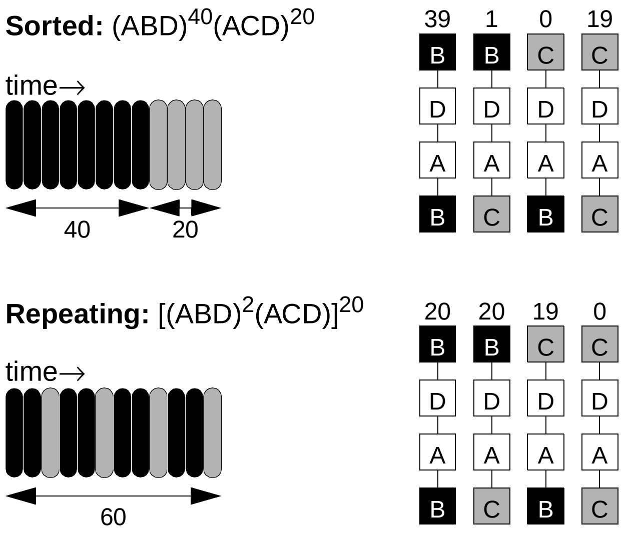


Fig. 6. Different possible behaviors for the example in Figure 3, shown graphically, as traces, and with the varying parts of the corresponding general path profiles.

repeating behavior patterns.

In the sorted case in Figure 6, the paths BDAB and CDAC have high execution frequencies, while the paths that switch sides of the loop, BDAC and CDAB, have low frequencies. In contrast, the repeating example has high frequencies for the switching patterns, a high frequency for path BDAB, and a frequency of zero for path CDAC. These differences are significant. If we can collect statistics about path frequencies, then some optimizations can exploit these differences in behavior to produce more efficient versions of the final program. In this article, we show how to exploit these differences to improve branch predictability. As another example, Young and Smith [1998] describe how to build a path-based loop unroller. This unroller makes copies of the loop body that track the paths with high frequencies.

Other researchers have investigated and implemented efficient forward path profilers [Bala 1996; Ball and Larus 1996]. The first six paths in Figure 5 are forward paths, but the four paths that distinguish our sorted and repeating examples will not appear in a forward path profile. Forward path profiles remain useful for capturing correlations within a loop iteration, but their exclusion of back edges prevents them from capturing cross-loop correlations.

2.3 Path Profiling

While paths may theoretically provide a useful compromise between point profiles and full program traces, they are not practically useful unless they can be efficiently collected. By "efficient" we mean both theoretically efficient under asymptotic analysis and practically efficient like other profiling and tracing techniques. In this section, we describe an asymptotically efficient algorithm for collecting path profiles of execution frequencies. We will return to the practicality of this algorithm later, in Section 4.2.

Ideally, the overhead of path profiling should be no higher than the overhead associated with point profiling or with tracing. Since there are more paths than points in a program, we expect the space requirements of path profiles to be greater than point profiles but less than the space requirements of a full trace. As for time efficiency, both point profiling and tracing methods incur overheads within a constant factor of the running time of the program, so we would want a path profiler to be similarly efficient. As we report later in Section 4.2, we find that our implementation of path profiling has a run-time overhead which is, in practice, comparable to the overhead of edge profiling. In all of our experiments, the overhead of path profiling is quite reasonable, and we have not spent the time to further optimize the process, though it is certainly possible to further reduce the overhead.

Any profiling algorithm must manage two sets of information: the data items and the descriptions of the program state where each datum was collected. A point profile requires space proportional to the size of the program (or more precisely, proportional to the number of points that were profiled). Descriptions of points in the program are compact: they may just be the index or address of the point in the program. A full program trace requires space proportional to the running time of the program. Paths are between points and traces: the program state is a path through the program, and there is one datum per path. Since the amount of space required is the product of the size of a state description and the number of such descriptions, some mechanism must be used to limit the amount of space required for both the descriptions of the paths and the per-path statistics.

In this work, we profile intraprocedurally (globally within a function, but not across function calls), and we place an upper bound on the number of edges that start at a split node. We call this upper bound the *history depth*; it is analogous to the number of bits in the history shift registers of dynamic branch prediction schemes. Using shorthand for history depth in formulas, we will sometimes use the variable k.[3] And for the sake of convenient notation, we define $k = 0$ to be

[3]The choice of k is an historical artifact. The hardware branch prediction schemes that inspired us to collect path profiles used k to describe the number of bits in a branch history shift register. This shift register approximates remembering the paths through the last k branches. A history

equivalent to edge profiling.

Using history depth as a bounding mechanism allows us to represent paths compactly: the bound on length ensures that a description of a path never takes more than $O(k)$ space. This bounding mechanism also limits the number of paths that we might encounter. Let P be the set of valid paths under our bounding mechanism. In the theoretical worst case, $|P| \leq |V| \times maxout^k$, where *maxout* is the maximum outdegree of any node in the CFG. This is because there are $|V|$ different possible nodes to start a path, and there might be $maxout^k$ different unique paths that start at each node. This gives a maximum storage requirement of $O(k \times |V| \times maxout^k)$ or $O(k \times |P|)$ space. In Section 4.2 we report the exact space requirements for our benchmark applications.

It is not clear *a priori* what is the best history depth to use for a particular algorithm. Our branch prediction transformation suffers decreasing marginal returns with deeper history depths. For a transformation such as this, the best strategy is to profile as deeply as practical, then design the transformation to use as much history information as is beneficial. Our transformation is designed in this manner; we use history depths up to 15 in this study without having deeply explored the marginal benefit from deeper or shallower history depths. It is an open research issue how to find the best history depth to use for a particular algorithm. It may also be useful to vary the history depth depending on the algorithm or the region of the program being executed; this is also open research.

Some comments about the paths induced by our bounding method are important for later discussion. Nodes or edges need not be unique in a path; multiple occurrences of a CFG node or edge merely indicate that the path covered a cycle in the original CFG. Legal paths include all paths up to and including the history depth of split nodes in length, so paths may share prefixes, suffixes, or subpaths.

To begin describing the profiling algorithm, assume that the algorithm has access to the trace of edges executed by the program. The trace might have been collected using a variety of efficient hardware or software mechanisms. The algorithm might run on-line (consuming the trace as it is generated) or off-line (reading the whole trace after it has been output), although it would be more desirable to have an on-line algorithm.

A naive way to collect path profiles would be to remember the most recent edges executed in a first-in, first-out (FIFO) queue as deep as the history depth. As each edge in the trace is examined, update the FIFO to include the new edge (possibly removing the oldest edge to comply with the history depth limitations), then look up and increment the counters for all of the paths represented by the edges in the FIFO. There is a small subtlety in this last phrase: we must update only the paths that start at old edges in the FIFO and reach all the way to the newest edge in the FIFO; otherwise we will overcount the subpaths represented in the FIFO (said another way, the paths that start at an old edge and reach to another edge that is not the newest were updated during an earlier state of the FIFO). Looking up a path will take $O(k)$ time.[4] There might be up to k paths represented in the FIFO,

depth of k is equivalent to remembering the directions of the previous k branches executed by the program.

[4]Even if you hash the paths, it still takes $O(k)$ time to examine the entire path.

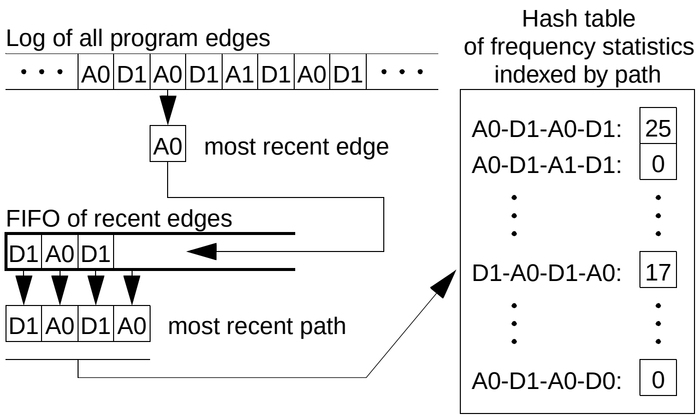


Fig. 7. A naive path-profiling algorithm. In this algorithm, the most recent edge in the program is shifted into the FIFO of recent edges, and the resulting path is looked up in the hash table. The statistic for that path is then updated. Edges are labeled by the name of their source block and whether the branch fell through ("0") or jumped ("1"). For correct statistics on shorter paths, we must also increment the counters for all subpaths ending with the most recent edge.

so this naive method would require $O(k^2)$ time per edge in the program. This naive method might be acceptable if we choose k to be sufficiently small. Figure 7 depicts this naive path-profiling algorithm.

Two observations allow us to do much better than $O(k^2)$ per branch in the program. First, it suffices to keep track of paths that are not suffixes of other paths, rather than all of the legal paths. With the frequencies from this "suffix-unique" set of paths, we can always infer the frequency of a shorter path. For example, if we know the frequencies of paths AXYZ, BXYZ, and CXYZ and know that these paths are the only paths that ended in XYZ, then the frequency of path XYZ must be the sum of the frequencies of the three longer paths.

In practical terms, the first observation allows us to speed up the naive implementation by one factor of k. We do not need to increment counts for paths in the FIFO other than the one that reaches from the oldest edge to the newest edge in the FIFO. All the other represented paths are suffixes of this longest path, so we can infer their frequencies in a postprocessing step. This means that we need to perform only one path lookup per edge, which takes $O(k)$ time per edge.

The second observation speeds up finding the next relevant path. Even with suffix-unique paths, there is still huge overlap between the paths considered in successive states of the FIFO. The $(k-1)$ suffix of the current path is the $(k-1)$ prefix of the next path. For example, if the current path is ABCD, then the next path must start with the prefix BCD. Further, the number of possible next paths is very small: it is the same as the number of successors of the most recent node in the current path. To continue the example, if X and Y are the only successors of D, then BCDX and BCDY are the only possible next paths. There is a successor relation between paths, which is a natural extension of the successor relation between nodes.

To exploit the second observation, we make a space-time trade-off. In addition to the description of each path, we store pointers to each successor path. This takes

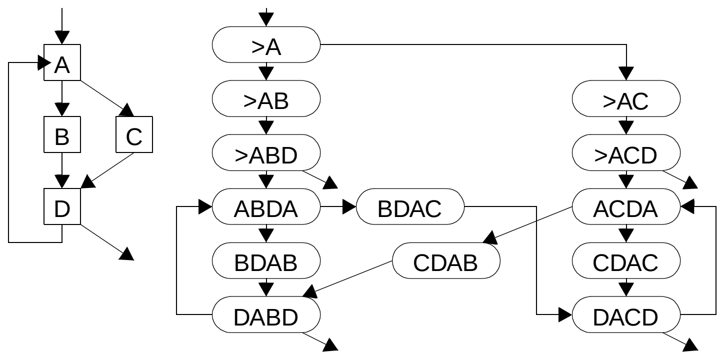


Fig. 8. The original CFG and its path CFG for paths of length up to 3. Note that paths have up to 3 edges and up to 4 nodes. A prefixed ">" before a path indicates a path that entered the original CFG from above block "A."

an additional $O(maxout \times |P|)$ space, for a total of $O((k + maxout) \times |P|)$ space. But then looking up the next path depends only on a table lookup based on the newest edge, so we need to perform only $O(1)$ work per edge in the trace. This is the same amount of work performed by point profilers and full program-tracing systems.

A third, practical observation can win back some of the lost space. We do not need to allocate the storage space for all paths before the program runs. Instead, we can create paths (and their pointers to other paths) lazily, as they are encountered dynamically. When we create a new path, we set all of its successor pointers to some null marker value (call it NULL, following C practice). If we find a NULL path successor pointer, then we need to either find or create the successor path. We can use a lookup method such as hashing to find whether or not the successor path already exists in $O(k)$ time ($O(k)$ to compute the hash value; $O(1)$ expected to search the hash table). If the successor path does not already exist, then creating it takes another $O(k)$ time. Then we update the pointer to the successor path.

This third observation requires further space and time analysis. The savings in space comes from lazy evaluation: we allocate space only for paths that were actually executed, rather than for the worst case. This remains $O((k + maxout) \times |P|)$, but $|P|$ may be much smaller than the exponential upper bound. The time analysis requires amortization. We still perform at least $O(1)$ work per edge in the trace, but we may also perform an additional $O(k)$ work when we find a NULL pointer. But the number of times we process a null pointer is small: it is at most $maxout \times |P|$. If the number of paths, $|P|$, encountered during a program run is much smaller than the number of edges in the trace, $|trace|$, then this additional work amortizes out to zero, so we still perform $O(1)$ work per edge in the trace. To be more precise about asymptotic analysis, point profiling and full program tracing require $O(|trace|)$ work during the course of running the program. The lazy version of our algorithm performs $O(|trace| + k \times maxout \times |P|)$ work during the course of running the program. If it turns out that $|P| \ll |trace|$, then the lazy algorithm is practically indistinguishable from point profiling or full tracing.

This efficient profiling method can be viewed as a lazy exploration of a graph that we call the *path CFG*. In the path CFG, each node is a path in the original CFG, and the successor relation is the natural path successor relation. The path CFG is much larger than the original CFG but is homomorphic to it. If we were to replace the nodes of the path CFG with code from the terminal block of each path, we would end up with a correct (but much larger) version of the original program, where each path in the original program was represented by a copy of its terminal basic block. Figure 8 depicts the path CFG for paths with length up to three from our example in Figure 3.

The algorithm presented efficiently collects profiles of path execution frequencies. It also has the desirable property that it can be run on-line, as it examines edges in the trace sequentially. The algorithm has comparable overheads to those of point profiling and full program tracing, as long as the number of paths is much smaller than the length of the trace. We will examine this assumption and explore the performance of an implementation of this efficient algorithm in Section 4.2.

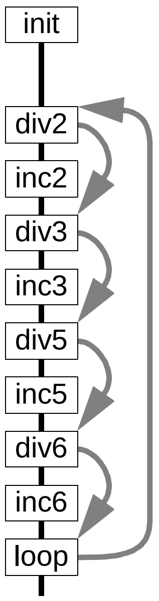
3. SCBP

This section describes *static correlated branch prediction* (SCBP), a code transformation that makes the conditional branches in a program more predictable. Predictability has an immediate benefit to performance, as modern processor designs perform better on programs with predictable control flow. Predictability can also have additional benefits during compilation: some optimizations make predictable programs run faster.

Our SCBP transformation consists of four major steps: profiling, local minimization, global reconciliation, and layout. We begin in Section 3.1 with a simple profiling example that we use to help explain the other steps in our algorithm. The next three sections present the other three major steps. The local minimization step examines the paths leading to each individual branch in the program to see if that branch correlates to the behavior of any of its CFG ancestors. Local minimization then determines the minimum amount of history necessary to achieve maximum prediction accuracy. The global reconciliation step combines the requirements of all branches in the program to produce a new CFG with the minimum number of basic blocks. The new CFG may have new basic blocks and edges in order to preserve history information and maximize prediction accuracy. The layout step chooses an ordering of the blocks in the new CFG, seeking to reduce the number of unconditional jumps in the program. After presenting the four major steps, Section 3.5 discusses how to tune the trade-off between improved prediction accuracy and code expansion to make SCBP improve performance on real programs. We tune the space-time trade-off by *overpruning* the history information from the second, local minimization step; overpruning discards marginal prediction accuracy in favor of better code size.

3.1 Path Profiling: A Simple Example

Path profiling was discussed in detail in Section 2. In this section we will introduce a simple example that helps to illustrate how SCBP works. Figure 9 depicts a program that loops through the integers from 1 to 100, counting up the numbers that are divisible by 2, 3, 5, and 6. In our benchmark suite, this is the *corr*



```
unsigned i=1, c2=c3=c5=c6=0;
do {
   if (i % 2 == 0)
      ++c2;
   if (i % 3 == 0)
      ++c3;
   if (i % 5 == 0)
      ++c5;
   if (i % 6 == 0)
      ++c6;
} while (i++ <= 100);
```

Fig. 9. The "corr" program: a simple program with branch correlation used to illustrate how SCBP works. "Fall-through" edges are shown in black without arrowheads, while "jump" edges are shown in gray with arrowheads.

microbenchmark.[5] Next to the program is its CFG, with the basic blocks named by abbreviations for what they do. Blocks with names like *"div2"* and *"div3"* test the variable i for divisibility by 2 and 3, respectively, then branch so that the appropriate increments of counters are performed. Blocks with names like *"inc2"* increment the matching counter, while *"loop"* tests the loop condition for the program. CFG edges that correspond to blocks with no branches or the fall-through path out of a conditional branch are drawn with simple lines; CFG edges that correspond to jumping to the target of a conditional branch are drawn grayed with arrowheads. Also, note that when i is divisible by a number, the branch condition will be true, but the branch will fall through so that the increment is performed; this may be the reverse of what one expects.

Basic arithmetic tells us that if i is divisible by 2 and 3, then it will also be divisible by 6. And we also expect that divisibility by 2 or 3 does not imply anything useful about divisibility by 5. If SCBP works correctly, then it should notice these cases and modify the program to exploit them. We expect that SCBP will make at least two copies of *div6* so that it can make one prediction for iterations where i was divisible by both 2 and 3 and a different prediction for iterations where i was not divisible by one of 2 or 3.

Figure 10 shows the paths whose last edge started at *div6* that were collected by

[5]We use this simple benchmark for pedagogical reasons. Though our benchmark suite in Section 4 does contain a variant of this program that iterates from 1 to 100,000, our results also show that we are able to uncover and exploit the correlation that exists in real-world applications.

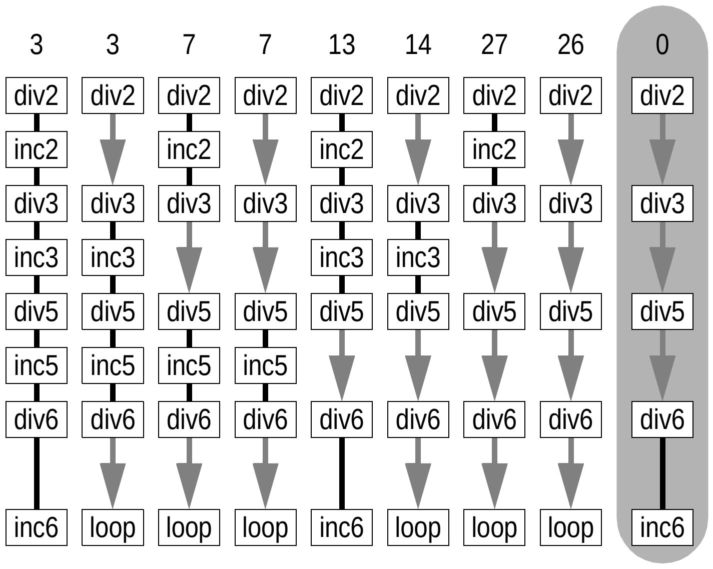


Fig. 10. Paths collected from running *corr* that help us to predict node *div6*. Each path's frequency is shown above it.

the path profiler during the run of *corr*. In other words, *div6* is the penultimate (second-to-last) block on these paths; we care not just about the path by which *div6* was reached, but also about which way the branch at the end of *div6* went when reached by that path. For the purposes of illustration, we use a history depth of 3 branches in this example. Note that a path corresponding to a depth of 3 branches contains 4 branches: the branch being profiled plus the preceding 3 branches of history. Furthermore, these paths contain a variable number of CFG nodes. The *"incX"* blocks do not end in branches, so they do not contribute to the history depth, although they do contribute to the length of the path.

Before continuing on to discuss local minimization, one observation is important. Some paths never occur because they are impossible: the rightmost path in Figure 10 would occur only for a number that was not divisible by 2, 3, or 5 but was divisible by 6. As we shall see, SCBP exploits paths with zero or low frequency to improve prediction accuracy.

3.2 Local Minimization

After collecting branch profiles, we examine each branch in the program to determine if its behavior correlates to any of its ancestors. If there is correlation, we also want to find the minimum amount of history information necessary to exploit that correlation. The less history information that we need to preserve, the less code expansion will result in the final program.

To look for branch correlations, we build a data structure called a *history tree* for each block in the program that ends in a conditional branch. Each history tree will be used to predict a single branch from the original program. At this point it helps to introduce some new terminology: when we build a history tree for a particular conditional branch, call that branch the *predicted branch* and call the basic block that it terminates the *predicted block*. Call any path whose last edge starts (not ends!) at the predicted block a *predictive path*. For each predictive path, call its last edge the *counted edge*, and call the rest of the path (the prefix of the path that omits the predicted edge) the *observed path*. By definition, all observed paths must end at the predicted block, and all counted edges must start at the predicted block. For a fixed observed path, there will be one predictive path for each counted edge out of the predicted block (predictive paths can have zero frequency). Then associate the frequency of each predictive path with the matching counted edge. These counts tell us the most likely successor of the predicted branch when it was reached by the observed path. Returning to the example in Figure 10, for the observed path *"div2* (jump) *div3* (jump) *div5* (jump) *div6,"* there are two predictive paths, one where *div6* falls through and one where *div6* jumps. The counts are 0 for falling through and 26 for jumping.

Predictive paths may seem unintuitive: we use them to predict their last edge, not their last block. We define predictive paths this way because a path tells us nothing about the exit edge from its final block.

A history tree summarizes common suffixes in observed paths (a history tree is a form of suffix tree). Each nonroot node in the history tree maps to an edge in the CFG; the root node of the history tree maps to the predicted block. Based on its position in the tree, each history tree node also represents an observed path in the CFG; this path is just the path found by taking the image of the path from the history tree node to the root of the history tree. Note that different nodes in the history tree may map to the same edge in the CFG, since there might be many paths from a CFG ancestor to the predicted block. For example, the edges exiting the *if* block of a conditional statement could appear multiple times in a history tree because the predicted block was reached along both sides of the conditional statement. Or a loop edge could appear multiple times in a history tree because path history covers more than one loop iteration.

Each node in the history tree represents a unique observed path; no two nodes in the history tree can represent the same observed path. More formally, the map from history tree nodes to CFG edges is not injective, but it induces an injective map from history tree nodes to CFG paths.

To make predictions, each node in the history tree holds a set of counters, one counter per counted edge from the predicted block. The counters record the frequencies for each of the counted edges that could be appended to the observed path to make a predictive path. In other words, the counters in a node show the frequencies with which each counted edge was traversed when the predicted block was reached by the observed path corresponding to that node. So if a predicted block has two exit edges, then every node in its history tree will hold two counters. The counters in the root node of the history tree show the total frequencies with which the predicted branch jumped or fell through. The counters in any other node of the history tree will have the same or smaller values than the counters in the root node; they show the frequencies with which the predicted branch jumped or fell through when it was reached by the observed path represented by that history tree node.

Figure 11 shows the history tree for block *div6*, which was assembled from the

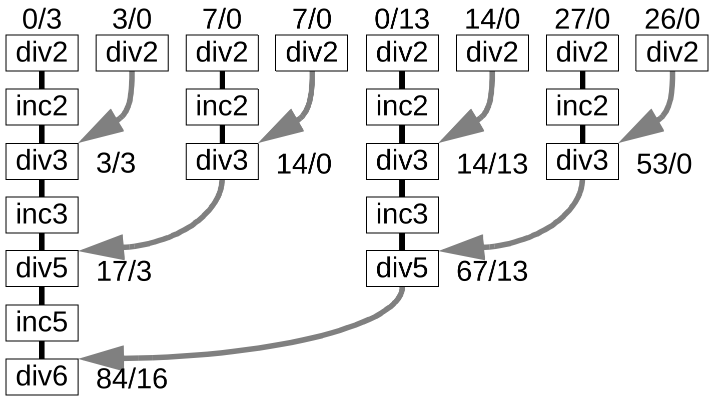


Fig. 11. History tree for block *div6*. "xx/yy" means that the execution reached basic block *div6* $xx + yy$ times along the path starting at that node, and that the branch at *div6* took xx of those times, while it fell through yy of those times.

paths depicted in Figure 10. We use graphics to indicate the kind of CFG edge represented by each history tree node. To represent the edge, "*div2* falls through," we draw a box labeled, "*div2*," with a fall-through (black, no arrowhead) line to its history tree parent. Similarly, the history tree node for the edge "*div2* jumps" is drawn as a box labeled *"div2"* with a jump (gray, with arrowhead) line to its parent. This allows us to draw history trees in a way that makes them resemble the CFG paths from which they are derived. But it is important to remember that history tree *nodes* represent CFG *edges*.

Notice how the path frequencies from Figure 10 have been rearranged and totaled into a pair of frequencies at each node (we omit counts for the *incX* nodes). In this example, one of the counts at each leaf node corresponds to the count from a nonzero path, while the other count corresponds to the count from a path with zero frequency. Each parent node totals the count from its children, summarizing them. Note that *div6*'s successors, *inc6* and *loop*, do not appear in the history tree; the edges leading to them are implied by the counts at each node. Also note that we draw history trees with the leaves at the top and the root at the bottom; this is botanically correct, but the reverse of traditional computer science practice. Lastly, note that edges in history trees (i.e., edges from parents to children) point in the reverse direction from CFG edges, even though our graphical conventions resemble the CFG edge from child to parent.

Figure 11 shows that correlation occurs. If SCBP can remember three prior branches worth of history (thus capturing divisibility by 2 and 3), then the direction of *div6*'s branch is completely determined by that history. This is shown in Figure 11 by the fact that the counts on all of the leaves of the history tree are of the form $0/yy$ or $xx/0$. Using correlation information, we can always predict *div6*'s branch correctly; simple static prediction based on point profiles would choose to predict the branch to be taken and be correct only 84% of the time.

To transform path profiles into history trees, we first partition the paths by their penultimate block. Each partition thus holds predictive paths for a single branch in

```
struct edge {          /* an edge in the CFG */
    int node_num;      /* source node's number */
    int succ_num;      /* which succ of the source node */
};

struct hnode {         /* history tree node data structure */
    edge mapto;        /* the edge to which this node maps */
    int *counts;       /* one per succ of predicted node */
    hnode *sib;        /* pointer to next sibling */
    hnode *kid;        /* pointer to first child */
};

struct hpath {         /* history path data structure */
    int freq;          /* execution frequency of this path */
    int length;        /* number of edges in this path */
    edge *edges;       /* array of edges in this path */
};
```

Fig. 12. Common data structure declarations for pseudocode in this article. Note that a pair of a node number and a successor number uniquely designates an edge in the CFG.

```
add_path_to_history_tree (hpath *p, hnode *n) {
    int last_edge = p->edges[p->length - 1]->succ_num;
    for (i = p->length - 1; i >= 0; i--)
n->counts[last_edge] += p->freq;
if some child c of n is the i'th edge in p then
    let n = c;
else
    create a new child of n that represents the i'th edge in p;
    let n = that new child;
}
```

Fig. 13. Iterative algorithm for adding a history path to a history tree. To build the history tree for a predicted branch, add all predictive paths for that branch to a starting node that maps to the predicted block.

the program. Then from the paths in each partition, we build the history tree for the common predicted block. For each path, we walk the path from the predicted node to the oldest node and simultaneously walk the corresponding nodes in the history tree. Along the way we augment the counters corresponding to the path's most recently visited edge by the path's frequency, and we create new nodes if necessary to represent the current path. This method ensures that parent nodes correctly accumulate counts for the common suffixes of the observed paths. This method for constructing history trees requires suffix-unique paths to work correctly; otherwise internal nodes may end up with counts that are not the sum of their children. Figure 12 gives common data structure declarations used in this article; Figure 13 gives pseudocode for the algorithm that builds history trees from predictive paths.

To find the minimum amount of history necessary to exploit correlation, we prune the history trees; the shape of the pruned tree will tell the global reconciliation step how much history to preserve. One can imagine a variety of degrees of pruning: pruning a tree all the way back to its root indicates that we wish to make a single per-branch prediction, while not pruning the tree at all indicates that every predictive path needs to be distinguished from every other predictive path (this

```
int prune_htree (hnode *nod) {
    int max_succ = -1, max_count = 0;
    for each counted edge e of the predicted branch do
if (max_count < nod->count[e]) then
    max_count = nod->count[e];
    max_succ = e;

    boolean children_agree = TRUE;
    for each child of nod c do
if (max_succ != prune_htree(c)) then
    children_agree = FALSE;

    if (nod is not a leaf and children_agree) then
prune all children;

    return children_agree ? max_succ : -1;
}
```

Fig. 14. Recursive pruning algorithm for history trees.

seems unlikely and would be very expensive in terms of code expansion). There is one mathematically natural way to prune: prune as much as possible without decreasing the potential branch prediction accuracy. And as we will discuss later, it is practically beneficial to prune further, sacrificing some prediction accuracy in order to limit code expansion.

In the example of Figure 11, the upper-right *div2* blocks (and their corresponding paths) have the same bias: it does not matter whether *i* was divisible by 2, because on both of these paths, *i* turns out not to be divisible by 3. Their bias agrees completely with the bias of their parent, so they can be pruned away. We can similarly prune the third and fourth *div2* blocks in the top row and the second and fourth *inc2* blocks in the second row; they also agree in bias with their parent. But we cannot prune any further without potentially losing prediction accuracy. The pruning algorithm is a simple bottom-up (leaves to root) recursive algorithm, where the children of a node may only be pruned if they all agree as to the most likely successor of the root node of the history tree. This pruning algorithm may lead to an uneven tree, but it always preserves a full tree where some sibling nodes on the frontier[6] always disagree (in a full tree, all nodes have either their original number of children or zero children). Figure 14 gives the algorithm for pruning history trees in pseudocode; Figure 15 shows the history tree from Figure 11 after running the recursive pruning algorithm.

In profiles with deep histories over programs with short loops, we would see nonroot nodes in the history tree that map to the predicted block. Such nodes, if not pruned away, indicate cases where the prior history of a branch relates to its future direction. If there is interesting correlation to exploit, SCBP may perform the equivalent of loop peeling or loop unrolling in order to capture the needed correlation history.

In many cases, we find that the history tree of a branch basic block prunes back to its root. This indicates that there was no global correlation visible for that branch

[6]The *interior* of a tree is the set of nonleaf nodes. The *frontier* is the set of leaf nodes.

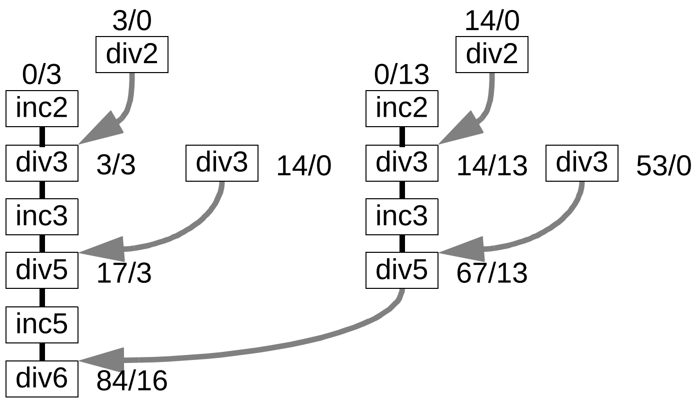


Fig. 15. Pruned history tree for block *div6*.

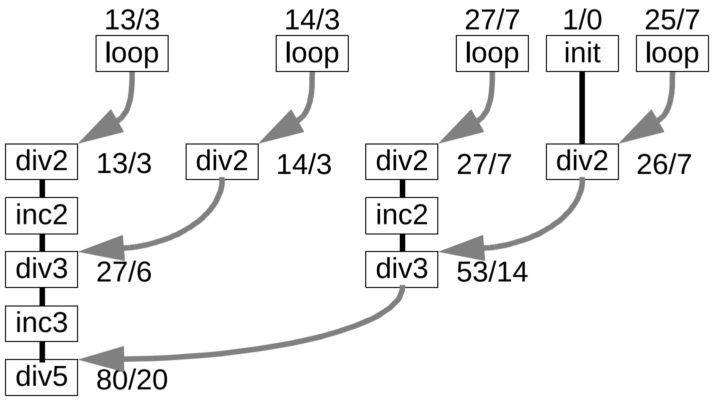


Fig. 16. History tree for block *div5* before pruning. In this case, recursive pruning will prune the tree back to its root, as all paths have the same bias.

in the profile. Figure 16 shows the history tree for block *div5*, where this is indeed the case. This corresponds to our intuition that there is no interesting correlation history for divisibility by 5.

3.3 Global Reconciliation

Having determined the minimal amount of history necessary to exploit correlation for each branch, the next step is to determine how many copies of each basic block must be made and how they must be connected so as to preserve correlation history. In a simple world, we would need at most two copies of each conditional branch basic block: one that predicts the branch will fall through, and one that predicts the branch will be taken. But predictions for many different future branches may need to be encoded into the program counter to maximize prediction accuracy. To do this, extra copies of CFG blocks may be needed to preserve correlation history information for these later branches in the program.

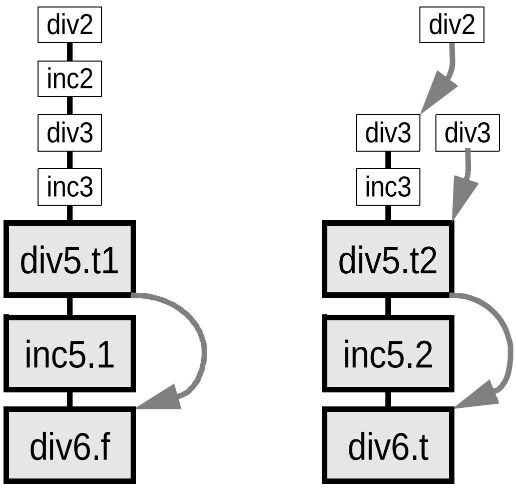


Fig. 17. Copies made of blocks *div5* and *inc5* during the global reconciliation step for *div6*. The large blocks show the actual blocks created; the small blocks show the execution history associated with the large blocks.

To illustrate the issues in reconciliation, consider block *div5*, whose history tree was pruned back to its root in the previous step. In a per-branch static prediction scheme, we would need just one copy of *div5*, say *div5.t* (the "*.t*" indicates that the branch in this copy is likely to be taken). But the path by which *div5* was reached turns out to be important for *div6*: if either *div2* or *div3* took (indicating a number not divisible by 2 or 3), then *div6* will be taken as well. With a single copy of *div5*, we cannot maintain a history of what happened during the execution of blocks *div2* and *div3* on behalf of *div6*, since we would traverse the single copy of *div5* in all executions.

In order to preserve history information for *div6*, we need to make multiple copies of *div5.t* on *div6*'s behalf. In particular, paths where neither *div2* nor *div3* were taken will connect to one copy of *div5* (call it *div5.t1*), while paths where *div2* or *div3* were taken will connect to a second copy of *div5* (call it *div5.t2*). Then we can connect all paths out of *div5.t1* to *div6.f* (the copy of *div6* that predicts that the branch in *div6* will be not taken) and all paths out of *div5.t2* to *div6.t* (the copy of *div6* that predicts that the branch in *div6* will be taken). Connecting copies this way ensures the correct static predictions for a given path history. Figure 17 shows each of these copies, along with the path history that each copy represents.

In the general algorithm, many more complicated splits may have to be performed. However, some splits may come "for free" because of work done for another node. In our example, if *div5* had a different branch condition so that it correlated the same way as *div6* to *div2* and *div3*, then *div5* would have needed two copies for different static predictions anyway. And *div6* would not have needed to further split *div5*'s copies, because the extant copies already preserved the necessary correlation history.

There are three major steps to the general global reconciliation algorithm. In the

```
for each basic block B in the program do
    let Split[B] = the set of splitters of B;

for each basic block B in the program do
    let Part[B] = the minimal set of copies of B;

for each basic block B in the program do
    for each successor T of B do
for each copy c of B in Part[B] do
    connect c to the appropriate copy of T in Part[T];
```

Fig. 18. High-level pseudocode for global reconciliation.

first step, we find all the potential *splitters* of each block in the original program. An interior node *b* in a history tree is a splitter of basic block B if it is necessary to have more than one copy of B so that the appropriate execution history can be maintained until we reach the block[7] corresponding to the root of the history tree. To distinguish between history tree nodes and basic blocks, we write history tree nodes in lowercase and basic blocks in uppercase. In the second step, we use the splitters to compute the minimum number of copies of each original program block. The copies will form the nodes of the output CFG. Finally in the third step, we examine each copy and use the history associated with the copy to connect it to the predecessor and successor copies that will maximize prediction accuracy. Connecting the copies in this third step creates the edges of the output CFG. Figure 18 shows the three major steps in the highest level of the reconciliation algorithm.

Finding splitters of a block occurs as follows: each nonleaf node of a minimized (pruned) history tree is a splitter of the source block of the edge to which it maps. A node might appear in multiple places in one history tree and in different history trees; we consider each occurrence of a node to be a different splitter. To find the splitters, we recursively walk through the history trees, assigning nonleaf nodes to sets of splitters. Figure 19 shows the pseudocode for this first step of global reconciliation. In Figure 15, the pruned history tree of *div6*, there are two occurrences each of *div3*, *inc3*, and *div5* and one occurrence each of *inc5* and *div6* in the interior of the pruned history tree; each of these nodes is a splitter of the blocks *div3*, *inc3*, *div5*, *inc5*, and *div6*, respectively.

The second major step in reconciliation determines the minimal number of copies of each basic block. Consider one basic block B from the original program. Let $Path[B]$ be the set of suffix-unique CFG paths that end at block B. The set of necessary copies of B is found by using the splitters of B to partition $Path[B]$. Each piece of the resulting partition requires one copy of B in the transformed program. Each piece is a set of paths; these paths should be the only paths that reach the corresponding copy of B in the output CFG. Frequently, the partition ends up with just one piece; this means that only one copy of B is needed in the output CFG.

Now consider one splitter, *b*, of basic block B. By definition, *b* was a nonleaf node

[7]Actually, when we are done, it may be a copy of the block corresponding to the root of the history tree.

```
void find_splitters () {
    for each block B do
let Split[B] = {};

    for each block B do
let T[b] be B's minimized history tree;
let b be the root of B's minimized history tree;
find_splitters_helper(b);
}

void find_splitters_helper (htree_node *n) {
    if n is not a leaf then
let N = n->mapto.node_num;    /* source of this edge */
add n to Split[N];
for each child c of n do
    find_splitters_helper(c);
}
```

Fig. 19. Pseudocode for finding splitters (part 1 of global reconciliation).

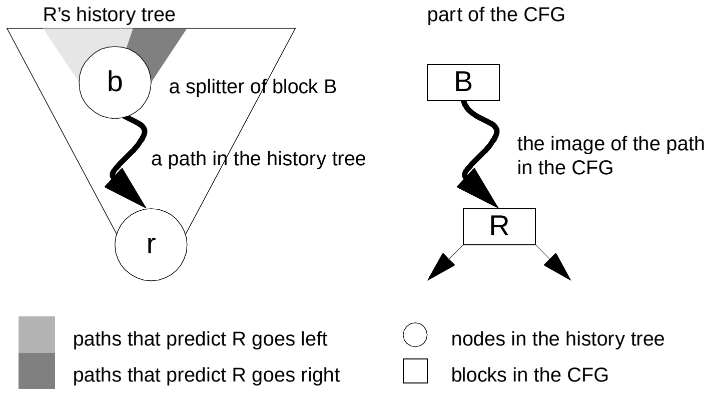

R's history tree
b
a splitter of block B
a path in the history tree
r
part of the CFG
B
the image of the path
in the CFG
R
paths that predict R goes left
paths that predict R goes right
nodes in the history tree
blocks in the CFG

Fig. 20. Drawing of definitions for the discussion of how splitters drive partitioning.

of some history tree where b was an edge that started at B. Splitter b is part of a history tree with some root node; call that root node r and call the basic block to which r maps R. Splitter b works on behalf of basic block R, possibly forcing additional copies of block B to be made so that R can be predicted accurately. By the way the pruning algorithm worked, we know that some children of b in R's history tree disagree about the prediction for R when R is reached by the image of the path from b to r. In plain English, at least two paths go through B and reach R but lead to different predictions for node R. These definitions are illustrated in Figure 20.

Let $T[b]$ be the history subtree rooted at b. Since b is a splitter, at least one path in $T[b]$ predicts one successor for basic block R, while some other path in $T[b]$ predicts a different successor for R. In fact, the frontier of $T[b]$ can be partitioned according to the successor of R that each path in the frontier predicts. In order for us to predict R correctly, there must be at least as many copies of B as there are pieces in this partition of $T[b]$. Each partition piece implies a different prediction for

R if R is reached by the image of the path from b to r. Further, this partitioning of $T[b]$ can be used to build a partitioning of $Path[B]$. Consider any path Π in $Path[B]$ (recall that Π is a suffix-unique path that ends at B). There is exactly one path π in $T[b]$ that maps to a suffix of Π. This fact allows us to build a function from $Path[B]$ to $T[b]$, where each suffix-unique path Π is assigned to its prefix in the history tree π. Then this function and the partition of $T[b]$ induce a partition of $Path[B]$.

There may be many splitters of each basic block in the program. Each splitter induces its own partition of the paths ending at that basic block. The intersection of these partitions is the final partition for that basic block. Call this final partition $Part[B]$. Since this is a finite intersection of partitions of a finite set, the final partition is well defined. To build the basic blocks of the output CFG, we create one copy of each original basic block for each piece of its final partition.

Implementing this intersection of partitions turns out to be a set of nested loops. We can consider each splitter individually, because applying the partition induced by a splitter monotonically refines the partition: once two paths have been put in separate pieces, they will never be reunited. To apply a splitter, we consider each piece of the partition so far, breaking pieces into finer pieces if required by the partition induced by the splitter. In practice, we find the piece of the partition induced by the splitter by tracing each path in $Path[B]$ along the corresponding path in $T[b]$. When we reach a leaf node in $T[b]$, we use the prediction of that leaf node to decide where to place the path. The pseudocode for the second part of reconciliation is shown in Figure 21.

The final partition piece that is associated with each copy is a set of paths; these paths are used in the third step of global reconciliation to determine the edges of the output CFG. It suffices to determine the successors of each block in the output CFG. Consider one piece of the final partition of basic block B; call it B1. B1 is a set of suffix-unique paths that end at B. B1 is associated with a copy of B in the output CFG; call this $Copy(B1)$. In the original CFG, B had some number of successors; we need to connect $Copy(B1)$ to copies of these successors. Let T be one of B's successors in the original CFG; without loss of generality, suppose that T was reached by the branch at the end of B jumping. Let e be the edge that connects B to T by jumping, and if Π is a path in B1, then let $S(\Pi, e)$ be the path obtained by appending e to Π, then possibly discarding some of the oldest edges in Π if necessary to preserve the history depth. Consider $S(B1, e)$, the image of B1 when we first extend each path in B1 using e, and then shorten to preserve history depth. Then we claim that $S(B1, e)$ is a subset of at most one piece of $Part[T]$. Assuming this claim is true, that unique piece corresponds to a unique copy of T in the transformed graph; this copy of T should be made the successor of $Copy(B1)$ by jumping. The pseudocode for connecting the different copies of the output CFG is given in Figure 22.

Our claim is stated precisely and proven in the following theorem:

THEOREM. *For each partition piece B1 in $Part[B]$, the final partition of block B, and for each successor T of B, where $e = (B, T)$ is an edge in the original CFG, $S(B1, e)$ is a subset of at most one partition piece in $Part[T]$.*

```
refine_partitions () {

    for each basic block B

/* Start with a single-piece partition */
let Part[B] = {{all suffix-unique paths ending at B}};

for each splitter b in Split[B]
    apply_splitter(Part[B], b);
}

apply_splitter (partition S, hnode *split) {

    let r = the root of split's tree;
    let R = r->mapto.source (split's predicted block);

    /* Subdivide each partition piece in S */
    for each piece Si of S

/* Need maxout(r) buckets for this splitter */
let N[0..maxout(r)-1] be empty sets;

for each path p in Si
    /* which way will R go when reached by p? */
    let predict = find_predict(split, p);
    add p to the set N[predict];

replace Si by nonempty pieces in N[0..maxout(r)-1];
}

find_predict (hnode *n, hpath *p) {

    for (i = p->length - 1; i >= 0; i--)
if n is a leaf
    return the index of n's largest counter;

some child of n must the same edge as the ith
edge in p, so let n = that child;

    /* never reached, by construction of htrees */
}
```

Fig. 21. Pseudocode for creating the basic blocks of the output CFG (part 2 of global reconciliation).

PROOF. By contradiction. Suppose it were otherwise: $S(B1, e)$ intersects with both T1 and T2, where T1 and T2 are different pieces of $Part[T]$, the partition of paths ending at T. For this to be true, there must be two paths Π1 and Π2 in B1, such that $S(\Pi1, e)$ is in T1 and $S(\Pi2, e)$ is in T2.

During the partitioning step, there must have been some splitter t of T that divided the paths in T1 from those in T2. In order to separate $S(\Pi1, e)$ from $S(\Pi2, e)$, t must have used some difference between them. But we know that the last edge in $S(\Pi1, e)$ and $S(\Pi2, e)$ was e, so $S(\Pi1, e)$ and $S(\Pi2, e)$ must differ at an earlier edge than e. This earlier difference must also be present in Π1 and Π2.

```
for each original block B in the program
    let Part[B] be the partition of paths ending at B;
    for each partition piece Bi in Part[B]
let C(Bi) be the copy associated with Bi;
for each edge e that starts at B
    let S(Bi, e) = {};
    for each path P in Bi
let tmp be the path obtained by appending e to P;
let S(P, e) be the path obtained by removing old
edges from tmp until |tmp| < history depth;
add S(P, e) to S(Bi, e);
    let T be the block reached by traversing e;
    let Part[T] be the partition of paths ending at T;
    let Ti be the piece of Part[T] of which S(Bi, e) is a subset;
    let C(Ti) be the copy of T associated with Ti;
    connect C(Bi) to C(Ti) using the same kind of edge as e;
```

Fig. 22. Pseudocode for connecting the blocks of the output CFG (step 3 of global reconciliation).

Since t was a nonleaf node, it must have a child node b that maps to e.

Claim. *t's child node b must be a splitter of B.*

Proof. By contradiction. Suppose b was a leaf node in r's history tree. We know that paths $S(\Pi 1, e)$ and $S(\Pi 2, e)$ both end in e. Using `find_predict` would return the same prediction for both paths. Then t would have assigned them to the same partition piece. But this contradicts our definition of t as the splitter that separated these two paths. □

We now know that Π1 and Π2 differ at some edge before e, and the claim showed that b is a nonleaf node in r's history tree. So b must assign them to different pieces of $Part[B]$. This contradicts our original assumption that Π1 and Π2 were in the same partition piece, B1. This contradiction proves the theorem. □

In a practical implementation of reconciliation, history paths with frequency zero are not considered during the partitioning step. Some nodes end up without connected successor edges because the counts for the corresponding counted edge for all observed paths leading to the node were zero. In such cases, we connect such edges to the most frequently executed successor copy.

Returning once again to our example, the CFG after reconciliation is drawn in Figure 23. As expected, there are two copies of *div6*. The first copy, *div6.f*, can only be reached if both *div2* and *div3* fall through. The second copy, *div6.t*, will be reached if either *div2* or *div3* takes. So SCBP performs as desired, creating the first two columns in Figure 23. It is also interesting to note the *div2.f* copy of *div2* in Figure 23. In the early blocks, *div2* and *div3*, a history depth of three allows SCBP to see backward past the loop block to the execution of *div6* in the previous iteration of the loop. The third column is reached only when *div6* takes, i.e., the value of i on the previous iteration was not divisible by 6. This means that two-fifths of the time i was even on the previous iteration, making it odd two-fifths of the time on the current iteration. This bias is used to predict that *div2* will fall through (indicating an even number) on this path.

The *div2.f* block of Figure 23 captures an interesting phenomenon: *statistical correlation*. While most compiler optimizations prove what must happen on all or

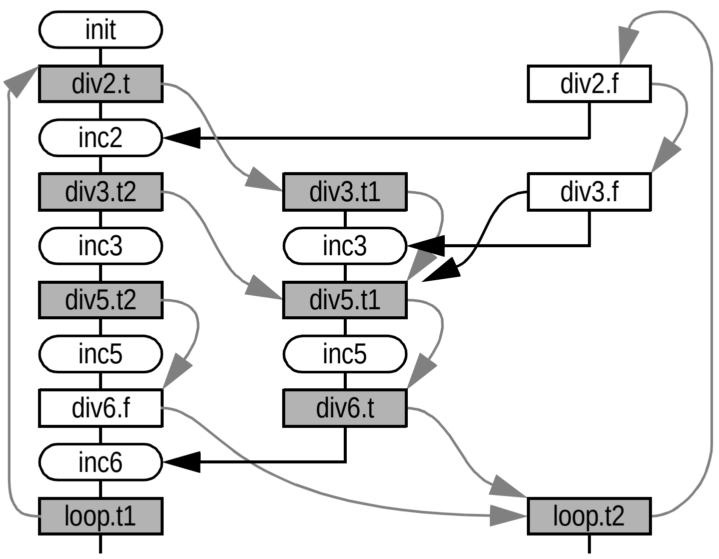


Fig. 23. CFG resulting from global reconciliation of the *corr* program. Shaded blocks predict that their terminal conditional branch will jump. While the original CFG in Figure 9 would have achieved a static branch prediction accuracy of 76.0% (120 mispredictions assuming all branches were predicted to be taken), this CFG achieves a prediction accuracy of 82.8% (86 mispredictions).

some paths of a program, the third column exploits something that is likely, but not certain, to happen. With a good optimizer, we might hope to find and exploit the logical correlation in the first two columns of Figure 23, eliminating a test for *div6* on some of the paths in the program. But a classical optimizer will never find statistical correlations.

The *div3.f* block of Figure 23 is a candidate for overpruning, a technique that we discuss extensively in Section 3.5. The bias of this branch is nearly 50%. The marginal benefit in terms of prediction accuracy is therefore small and probably much less than the cache costs of this code duplication.

3.4 Layout Issues

The CFG produced by reconciliation typically is not an executable program, because reconciliation may have created new join points. If SCBP is performed in an intermediate representation where control-flow is represented by graphs, new join points do not matter. But in object code, or in intermediate representations that resemble machine instructions, branches may have only one explicit target. New join points may require additional branch instructions in order to ensure correct program semantics. Transforming a control-flow graph into a linear sequence of instructions is called the *code layout* problem; algorithms that attack code layout attempt to reduce cache misses or pipeline stalls in the program by changing the order of basic blocks or procedures [Calder and Grunwald 1994; Hwu and Chang 1989; McFarling 1993; Pettis and Hansen 1990; Torellas et al. 1995; Young et al. 1997].

In the first conference paper on SCBP [Young and Smith 1994], we attempted to improve branch prediction accuracy without increasing instruction count. To do this, we used a simple and inefficient layout algorithm that copied code until an unconditional branch or the end of a procedure was reached. This algorithm further increased code expansion due to SCBP, but it allowed us to exhibit branch prediction gains and pessimistic code expansion. However, there are a number of effective existing code-layout schemes that insert a small number of unconditional branches instead of duplicating code, and it makes sense for us to use one of them.

Pettis and Hansen [1990] described a simple, greedy, profile-driven scheme for basic-block placement in their paper on code placement optimizations. In brief, they sort the CFG edges by profiled frequency, and then they lay out the CFG by connecting edges whenever possible in sorted frequency order. This greedy heuristic attempts to minimize the number of taken branches in the code; whenever a high-weight edge is placed as a taken branch, it must have been the case that some higher-weight edge had already connected to the same destination, or that the taken branch is necessary to close a loop. Calder and Grunwald [1994] proposed a more refined greedy approach, and we [Young et al. 1997] have done work on nearly optimal code layouts, but for the purpose of this work, we use Pettis and Hansen's greedy method to lay out the CFG.

3.5 Trading Off Space and Time

As described so far, SCBP attempts to capture the maximum improvement in prediction accuracy with no attempt to limit code expansion. But the net effect on performance will depend both on improvements due to better branch prediction and on penalties due to worse cache miss rates. As SCBP duplicates code, it will reach a point of negative returns, where the marginal benefit from improved prediction accuracy is more than outweighed by the marginal penalty from worse cache miss rates. Rather than making the maximum number of copies to achieve the maximum improvement in prediction accuracy, it will be better to choose only the most profitable branches and blocks for duplication.

Figure 24 depicts a case where unconstrained SCBP performs poorly. It shows a code fragment from the *massive_count* procedure in the *espresso* benchmark. This part of the *massive_count* routine updates counts of the bits that are set in a bit vector. Each increment of a count is guarded by a branch that tests the corresponding bit in the bit vector. These guards form a sequence of *if-then* branches. The way that SCBP works, it will attempt to predict each bit in the vector based on the preceding bits in the vector. But in general we expect to find very little correlation between neighboring bits in bit vectors (and even less correlation across different training and testing data sets), while the code expansion is potentially exponential in the history depth. To make SCBP practical, we need to find a way to avoid making exponential copies of branches in this part of *massive_count*, while still exploiting useful correlation in other parts of the program.

Overpruning the minimized history trees gives us precise fine-grain control over the trade-off between prediction accuracy and code expansion. While we called the result of the second step a "minimized" history tree, this minimization was with respect to prediction accuracy, not code expansion. A minimized history tree preserves the minimum path history necessary to perform prediction with the same accuracy as the original, unpruned history tree. By pruning even farther (i.e.,

```
if (val & 0xFF000000) {
    if (val & 0x80000000) cnt[31]++;
    if (val & 0x40000000) cnt[30]++;
    if (val & 0x20000000) cnt[29]++;
    if (val & 0x10000000) cnt[28]++;
    if (val & 0x08000000) cnt[27]++;
    if (val & 0x04000000) cnt[26]++;
    if (val & 0x02000000) cnt[25]++;
    if (val & 0x01000000) cnt[24]++;
}
```

Fig. 24. A worst-case code fragment for unconstrained SCBP. Bits in `val` are unlikely to be correlated, but SCBP will duplicate code to exploit minor variations in frequency on each path through the sequence of tests. Further, the minor frequency variations are unlikely to remain across data sets: this is false correlation. In the worse cast, examples like this can lead to exponential code expansion and no actual improvement in prediction accuracy.

overpruning), we can sacrifice prediction accuracy to limit code expansion.

For this study, we implemented a simple overpruning heuristic that sorts splitters by their marginal benefit, then selects splitters until most of the benefit from correlation has been captured or until the marginal benefit of the next splitter has decreased by some limiting factor below the marginal benefit of the most valuable splitter. This overpruning heuristic is blind to specifics such as cache size and program size that might allow a better trade-off to be made. But as we will see in Section 4, this heuristic works well enough for most of our benchmarks.

Determining the marginal benefit of a splitter is simple: just subtract the best possible prediction for the split node from the sum of best possible predictions for its children. This difference represents the improvement in prediction accuracy that would result from making a prediction for each path from a child of the splitter to the root of the history tree versus making a single prediction for the path from the splitter to the root of the history tree. After sorting the splitters by marginal benefit, we iterate from most beneficial to least beneficial splitter, marking each splitter as we reach it. We stop marking splitters once we accumulate 50% of the total possible improvement in prediction accuracy or if the benefit due to the next splitter is less than one tenth of the benefit due to including the most beneficial splitter.[8]

After we have marked the most useful splitters, we then run a recursive overpruning step. The overpruning step prunes any node whose parent has no marked descendants. Pseudocode for all of the overpruning step is given in Figure 25; we add two new fields, `mark` (a boolean) and `benefit` (a frequency count), to each history tree node to implement overpruning.

After overpruning, global reconciliation and layout operate without modification. This claim requires some justification for the reconciler, because overpruning might be imagined to change the partitions in such a way that connecting the edges of the transformed CFG would no longer work. This is not the case because overpruning is still a form of pruning: any split performed for a child history tree node will have a corresponding split in the parent history tree node. The parent split ensures that there will always be at most one partition piece that matches the connection

[8]These heuristic values were chosen as reasonable guesses. No effort has been made to tune them.

```
/* Add these new fields to struct htree: */
    boolean mark;
    int benefit;

over_prune_step () {
    total_benefit = 0;
    for each splitter s
let s->benefit = max_count(s);
for each child c of s
    s->benefit -= max_count(c);
total_benefit += s->benefit;

    sort splitters by benefit;

    cumulative_benefit = 0;
    first_benefit = 0;
    for each splitter s from largest to smallest benefit
if (first_benefit == 0)
    first_benefit = s->benefit;
if (cumulative_benefit > 0.50 * total_benefit
     || benefit(next_splitter) < 0.10 * first_benefit)
    break;
s->mark = TRUE;
cumulative_benefit += s->benefit;

    for each history tree h
over_prune_htree(h);
}

max_count (htree *n) {
    return the maximum value in n->counts;
}

over_prune_htree (htree h) {
    boolean stop = h->mark;
    for each child c of h
stop |= over_prune_htree(c));
    if (!stop)
prune all of h's children;
    return stop;
}
```

Fig. 25. Pseudocode for the overpruning step.

criteria for the third stage of reconciliation. Alternatively, one can state that the proof of reconciler correctness relied only on the relationships between history tree nodes and never used minimality of the history trees, so the proof still applies to nonminimal history trees.

4. EXPERIMENTAL RESULTS

This section measures the practical quantities associated with our general path profiling algorithm and with the SCBP optimization. Section 4.1 begins by describing our experimental benchmarks, compiler, test machine, and measurement methodology. Section 4.2 measures the space and time overheads from a simple

Table I. Benchmarks Used in this Study

Category	Benchmark	Description
microbenchmarks	alt	Sorted example
	ph	Repeating example
	corr	Branch correlation example
	wc	UNIX word count program
SPECint92	compress	Lempel/Ziv file compression utility
	eqntott	Translates boolean eqns to truth tables
	espresso	Boolean function minimizer
SPECint95	gcc	GNU project C compiler
	go	Plays the game of Go
	ijpeg	JPEG encoder
	li	XLISP interpreter
	m88ksim	Microprocessor simulator
	perl	Interpreted programming language
	vortex	Object-oriented database

implementation of the path profiling algorithm of Section 2. Section 4.3 reports our simulation results for branch prediction accuracy and cache miss rates of programs transformed by SCBP coupled with Pettis-and-Hansen-style code layout. Section 4.4 analytically examines the performance of SCBP, while Section 4.5 reports wall-clock times for SCBP-transformed programs on actual hardware.

4.1 Benchmarks and Methodology

We used the benchmark programs listed in Table I to evaluate the performance benefit of SCBP. The first three programs are microbenchmarks. They are idealized examples of the kind of behavior that can be exploited using path profiles but is invisible using point profiles. The fourth program, *wc*, is frequently used for studies in instruction-level parallelism and can be thought of as a microbenchmark. Besides these, we mainly use a number of SPECint benchmarks from the 1992 and 1995 suites. The SPECint benchmarks tend to have shorter basic blocks and more unpredictable control flow than the SPECfp benchmarks. We include *compress* from the 1992 suite because it runs for fewer cycles than the 1995 version but is substantially the same program. We include *eqntott* from the 1992 suite because it is frequently used to illustrate branch correlation in the research literature; we include *espresso* from the 1992 suite because it includes a worst-case example for the SCBP algorithm of Section 3.

To fairly assess a profile-based technique such as SCBP, we use *cross-validation*: an application is optimized using the profile data gathered by running the application on a training data set, and then results are reported for running this optimized executable on a separate testing data set. Table II lists the training and testing inputs for each benchmark. The microbenchmarks, unlike the application benchmarks, take no input; we simply list "null" as their training and testing inputs.

Fisher and Freudenberger [1992] performed an extensive cross-validation study on profiled static branch prediction, concluding that good training sets for static branch prediction can be found. We have not done an extensive cross-validation study to show that optimizations using path profiles are robust across a wide variety of training data sets. Such a cross-validation study is open research; it is not even clear that path profiles are mathematically less robust than point profiles, because

Table II. The Input Data Sets Used with Our Benchmarks.

Benchmark	Training Data Set	Testing Data Set
alt	null	null
ph	null	null
corr	null	null
wc	compress92 ref input	PostScript for a conference paper
compress	compress92 ref input; source code	MPEG movie data (6MB)
eqntott	fixed to floating-point encoder	priority encoder, SPEC92 ref input
espresso	ti, part of SPEC92 ref inputs	tial, part of SPEC92 ref inputs
gcc	amptjp.i, SPEC95 train input	cccp.i, part of SPEC95 ref inputs
go	2stone9, SPEC95 train input	9stone21, part of SPEC95 ref inputs
ijpeg	vigo, SPEC95 train input	vigo, SPEC95 ref input
li	SPEC95 train input	SPEC95 ref input
m88ksim	dcrand, SPEC95 train input	dhry, SPEC95 test input
perl	primes, SPEC95 train input	primes, SPEC95 ref input
vortex	SPEC95 train input	SPEC95 test input

Table III. Additional Information about Benchmark Runs. "Size" is the number of bytes in the dynamically linked binary executable file of the original version of the program. For the training and testing data sets, "M Branch" counts the millions of branches executed, while "M Instr." counts the millions of instructions executed by the original version of the program.

		Training Data Set		Testing Data Set	
Benchmark	Size (KB)	M Branch	M Instr.	M Branch	M Instr.
alt	25	-	-	2.0	12.3
ph	25	-	-	2.0	11.3
corr	25	-	-	5.0	2.2
wc	25	6.6	26.1	14.6	60.2
compress	57	11.8	112.1	135.4	1,305.9
eqntott	115	46.5	388.0	335.8	2,519.8
espresso	565	87.0	745.5	157.2	1,247.8
gcc	5,595	252.6	2,015.1	244.1	1,941.0
go	918	68.1	738.0	4,177.3	45,612.9
ijpeg	573	97.4	2,451.5	1,801.3	51,934.4
li	279	34.3	337.9	10,961.5	81,323.2
m88ksim	532	18.5	203.1	116.0	943.1
perl	1,032	1.4	12.5	2,274.6	20,170.3
vortex	1,737	292.0	3,690.0	1,068.8	13,208.6

point profiles are often used by optimizations to simulate path statistics.

Table III lists the branch and instruction counts for each of our benchmarks under each data set. Branch counts were collected using branch instrumentation, while the instruction counts were collected using a compiled simulation of a basic-block-scheduled version of each program. The major point of Table III is that our benchmarks execute a significant number of instructions and are reasonable to use in branch prediction studies.

For all of our experiments, we used the SUIF compiler system (version 1.1.2) from Stanford [Wilson et al. 1994], including Machine SUIF (version 1.1.2) extensions from the HUBE research group at Harvard [Smith 1996]. SUIF is a research compiler that has been used for studies in parallelization, memory alias analysis, and interprocedural loop detection. The Machine SUIF extensions support machine-specific optimizations and have been used for studies in branch prediction, code layout, register allocation, and instruction scheduling. For profiling, we used

HALT [Young and Smith 1996], a profiling package that works with Machine SUIF.

We use the efficient profiling method described in Section 2 within HALT to insert profiling code into our programs. HALT works similarly to Atom [Srivastava and Eustace 1994], an instrumentation tool that inserts calls to user-written analysis routines into a program. The HALT and Atom approach incurs a function call overhead at each instrumentation point, but offers tremendous flexibility in the kind of analysis that can be performed. More special-purpose profilers have achieved much lower profiling overheads [Ball and Larus 1994; 1996], but are often tailored to specific kinds of profiling and machine implementations. For our purposes, $O(1)$ overhead is asymptotically the same as traditional branch-profiling methods, so we have not taken the time to build a more efficient profiler.

Simulations, compilations, and performance timing experiments were performed on an AlphaPC 164LX with an Alpha 21164 microprocessor running at 533MHz. Profile timing experiments were performed on an AlphaStation 500 with an Alpha 21164 microprocessor running at 266MHz. Both of these Alpha workstations have 2MB of third-level off-chip combined instruction and data cache and 128MB of main memory. The 533MHz workstation ran Digital UNIX 4.0D; the 266MHz workstation ran Digital UNIX 4.0.

4.2 Overheads in Path Profiling

This section describes and measures the performance of an implementation of the path-profiling algorithm from Section 2.3. We implemented the algorithm under the SUIF compiler, using Machine SUIF extensions and the HALT profiling environment. HALT provides CFG edge tracing, so our implementation of the path-profiling algorithm was relatively straightforward. A labeling pass assigns a unique number to each branch in the program. The Machine SUIF CFG library [Holloway and Young 1997] provides an abstraction of a program as a graph where the successors of a node are sequentially numbered starting at zero. Each edge can then be uniquely described using the pair of the unique branch number and its successor number.

The implementation represents paths compactly, storing just the starting node of the path and the sequence of successor numbers from subsequent branch nodes in the CFG. Since the path stores only branch nodes, a path of length n may include more than n CFG nodes. Successor numbers are concatenated into a packed bit representation; successor numbers may only be accessed by sequentially shifting and masking numbers off this packed representation. We have run with history depths up to 20 without overflowing the 64-bit registers on the Digital Alpha microprocessors used in this study.

Very little tuning has been done on the implementation of our algorithm. As noted previously, HALT incurs a function call overhead at every instrumentation point; this is every call, return, conditional, or multiway branch in the program. HALT ensures safety by computing the live registers of the program at each instruction, then saving all live registers at the instrumentation point. It then collects any desired run-time information (e.g., successor numbers in our case), then passes that information and any compile-time-specified constants (e.g., unique branch number) to the analysis function. Inlined profiling code would certainly be faster. Also, the path data structure is implemented as a C++ class, and the layers of storage for

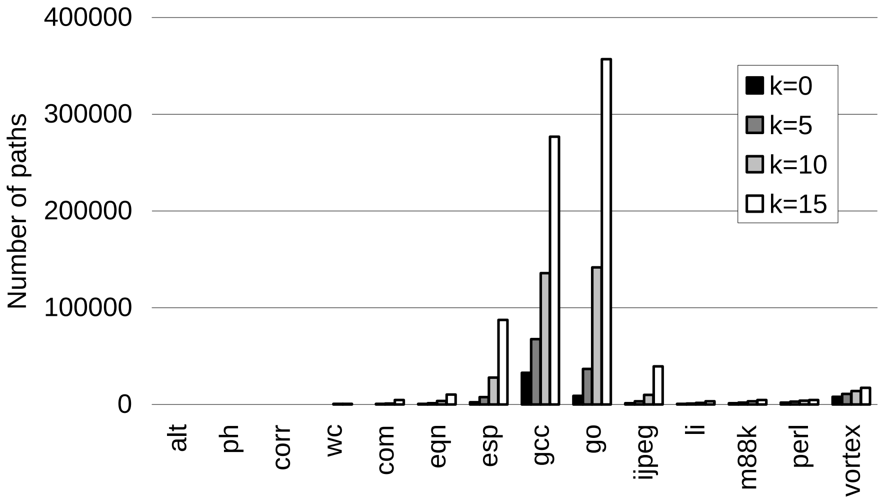


Fig. 26. Number of paths profiled as a function of history depth.

inheritance cause a large amount of data structure bloat. As we are about to see, the number of paths we collect and the running time of the algorithm have been acceptable, so we have not spent additional time tuning the profiling step.

Acceptable performance requires good behavior in both space and time. In practical terms, we would like the path profiles to take up much less space than a full program trace (however, we expect that path profiles will include more samples than point profiles). And we would like the time for path profiling to be within a small constant factor of point profiling.

Figure 26 graphs the number of paths collected as a function of the history depth. The number of paths collected is fairly small in most of the benchmarks; this is consistent with the findings of Ball and Larus [1996]. The worst offenders in terms of number of paths are the *gcc* and *go* benchmarks. *gcc* reaches a maximum of 276,758 paths at a history depth of 15, while *go* reaches a maximum of 356,860 paths at a history depth of 15. These are not overly large databases of information for a compiler to handle.

Figure 27 shows the same data as Figure 26, but normalized against the number of points in an edge profile ($k = 0$). Recall from Section 2.3 that the worst-case number of paths is exponential in the history depth. Figures 26 and 27 show that the relationship between history depths and the number of paths is indeed exponential, but that for the benchmarks in this study and the history depths we consider, the number of paths encountered is not overwhelmingly large. It is interesting to note that benchmarks with large absolute numbers of paths (high peaks in Figure 26) are not necessarily the same benchmarks with large normalized numbers of paths (high peaks in Figure 27). In particular, *wc* and *compress* have small absolute numbers of paths, but because the branches in these benchmarks are mostly data-driven, the small number of branches contribute to large normalized numbers of paths.

Figures 26 and 27 show that the number of paths can be exponential in the history depth. This is a fundamental property of path profiles. Heuristics that constrain

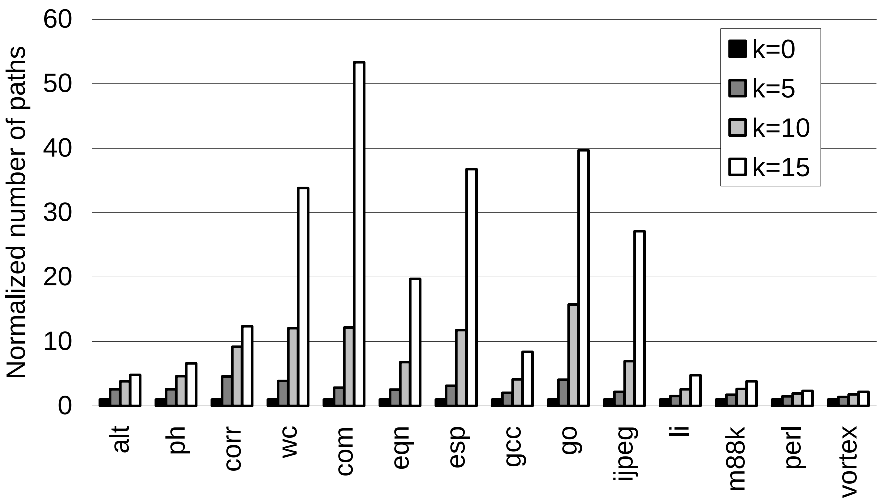


Fig. 27. Number of paths profiled as a function of history depth, but normalized against $k = 0$ (edge profiling).

the number of paths or the history depth in infrequently executed parts of the program may help to limit the number of paths collected, but they probably will not change the exponential number of paths, as most paths occur in the frequently executed parts of the program. Since the number of paths is not unmanageably large at the history depths we explore, we have not investigated any space-saving heuristics.

To show that our implementation runs in reasonable time, Table IV lists and Figure 28 graphs running times of our benchmarks under various instrumentation schemes. The “Original” column shows the running time of the unmodified program. The “Null Analysis” column gives the running time of the program modified by HALT to call empty analysis functions that return immediately. The difference between “Original” and “Null Analysis” corresponds to the overhead introduced by HALT-style instrumentation alone. The “Point Profile” columns show the running time of node and edge profilers under HALT; the “Path Profile” columns show the running time of our efficient path profiler with history depths of 0 (equivalent to edge profiling) and 15. Times were collected using the UNIX `time(1)` command on a multitasking machine; factors like multitasking activity or buffer cache status may have influenced the results. We were less precise about timing our profiling versions of the program because the point of Table IV is to show general ranges of performance rather than to make precise statements about the performance of untuned profilers.

From Table IV, one can see that the “Point Profile” and “Path Profile” columns are often close. For most benchmarks the difference is never more than a factor of 2; *go* is the sole exception, where path profiling takes 4.1 times longer than edge profiling. Compared to the original program, the worst case for point profiling is *wc* (which is 12 times slower than the original program), while the worst case for path profiling is again *go* (which is 30 times slower than the original program).

Table IV. Profiling overheads under HALT. Times are for the training data sets.

Benchmark	Time (seconds)					
		Null	Point Profile		Path Profile	
	Original	Analysis	Node	Edge	k=0	k=15
alt	0.2	0.9	0.7	0.9	0.9	0.9
ph	0.1	0.3	0.6	0.8	0.8	0.8
corr	0.2	0.2	0.3	0.3	0.3	0.4
wc	0.2	0.8	1.8	2.4	2.5	2.5
compress	1.3	2.2	4.1	5.6	5.4	7.6
eqntott	3.6	7.9	15.9	22.0	21.9	26.8
espresso	4.0	14.3	29.9	37.9	39.6	67.5
gcc	19.2	53.7	104.8	140.8	167.4	270.2
go	5.2	14.2	27.9	38.0	42.3	154.7
ijpeg	11.1	21.4	38.5	47.6	48.4	66.5
li	2.0	6.2	13.6	17.2	19.9	21.1
m88ksim	1.6	4.4	8.6	11.4	11.6	13.3
perl	0.6	1.1	1.5	1.9	1.8	3.4
vortex	25.9	65.4	160.6	219.5	210.1	209.8

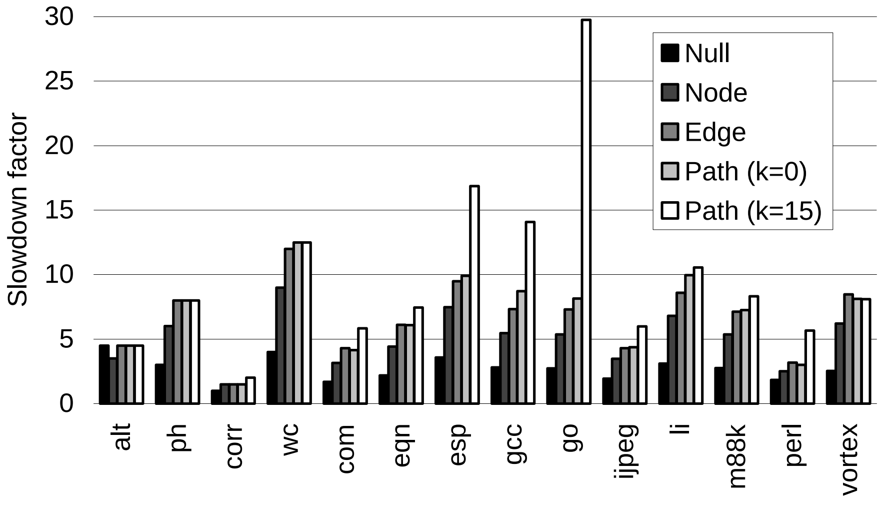


Fig. 28. Profiling Overheads under HALT. A slowdown factor of 1 corresponds to the running time of the original program.

4.3 Prediction Accuracy and Cache Miss Rate

Tables V and VI and Figures 29 and 30 show the effect of SCBP on program size, branch misprediction rate, and cache miss rate. These statistics were collected by transforming each program using SCBP (with $k = 15$ and overpruning) and Pettis-and-Hansen-style basic-block and procedure placement. Table V displays the size of the text segment of each program as determined by the UNIX `size` command.

Table V. Static Code Size of Original and Transformed Programs.

	Size in bytes		Percent
Benchmark	orig	scbp ($k = 15$)	increase
alt	4104	4328	5.46
ph	4008	4056	1.20
corr	4248	4472	5.27
wc	4328	4328	0.00
compress	18640	19280	3.43
eqntott	45888	46016	0.28
espresso	243368	247704	1.78
gcc95	2266776	2268728	0.09
go95	412368	420992	2.09
ijpeg95	241864	263736	9.04
li95	106280	107224	0.89
m88ksim95	210240	210304	0.03
perl95	489704	490216	0.10
vortex95	747304	748872	0.21

Table VI. Mispredict Rates and Cache Miss Rates for Programs Transformed with Overpruning SCBP. Static branch predictions were made by SCBP. Pettis-and-Hansen-style block and procedure placement was performed. If a benchmark has both a training and a testing data set, then the results shown are cross-validated by profiling and optimizing using the training data set then running on the testing data set. If not (*corr*, *alt*, and *ph*), then the results shown are from training and testing on the same data set.

	Mispredict Rate (%)		Cache Miss Rate (%)	
Benchmark	orig	scbp ($k = 15$)	orig	scbp ($k = 15$)
alt	12.5	0.0	0.0	0.0
ph	15.0	0.0	0.0	0.0
corr	24.0	12.7	0.0	0.0
wc	14.4	8.8	0.0	0.0
compress	14.5	12.1	0.0	0.0
eqntott	13.4	12.2	0.0	0.0
espresso	20.3	18.7	0.1	0.1
gcc	12.8	12.5	1.6	1.6
go	22.4	20.4	0.9	1.5
ijpeg	11.5	9.8	0.0	0.1
li	14.7	12.9	0.2	0.1
m88ksim	9.0	8.9	0.2	0.2
perl	11.0	10.1	0.3	0.4
vortex	0.9	0.8	2.0	1.1

None of our programs increase in text size by more than 10%.

We then instrumented each transformed program to collect branch prediction statistics and to perform a cache simulation. The "orig" column shows results for per-branch static prediction, while the "scbp" column shows results for SCBP run on path profiles. Static predictions were set by SCBP using the training data set; for all benchmarks with both training and testing data sets we show mispredict rates for runs using the testing data set. The cache simulator simulated a 32-kilobyte, direct-mapped cache with a 32-byte line size; this size is slightly larger than the caches found in microprocessors that are shipping today, so these cache results will be optimistic compared to what we might expect on real processor implementations.

All of the SPEC benchmarks show improvements in prediction accuracy after transformation using SCBP; the arithmetic average over SPEC benchmarks is an

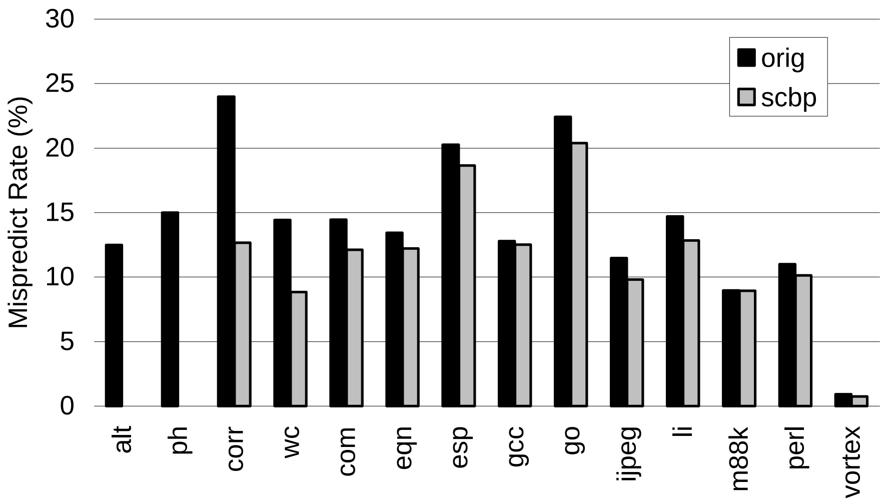


Fig. 29. Graph of the effect of SCBP on branch mispredict rate. Data are taken from Table VI.

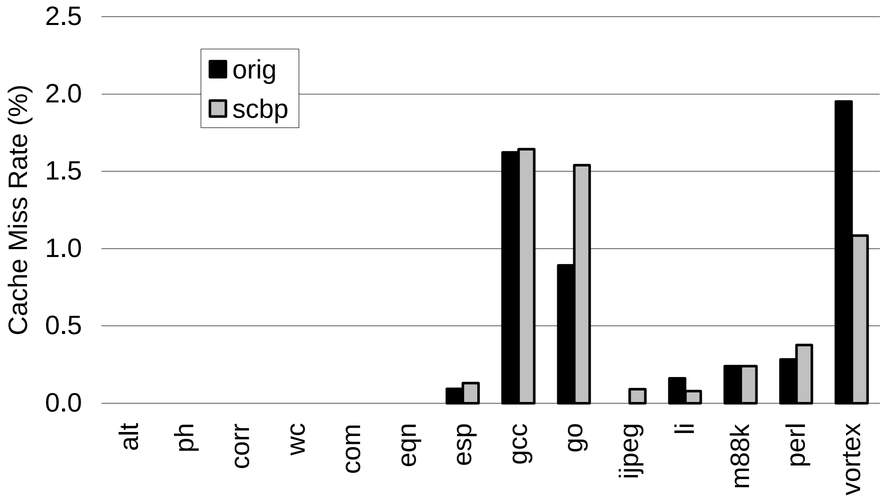


Fig. 30. Graph of the effect of SCBP on cache miss rate. Data are taken from Table VI.

improvement of 1.2%. Cache miss rates are interesting for the SPEC95 benchmarks. The *go* benchmark has bad cache behavior under SCBP (its working set is much larger than 32KB); in this case an overpruning heuristic that took cache size and program working set into account would restrict code expansion much more drastically. *espresso*, *gcc*, *ijpeg*, and *perl* also suffer from increasing cache miss rates under SCBP. *Li* and *vortex* each *improve* in cache miss rate under SCBP; this is due to variations in procedure placement in the cache.[9] Unsurprisingly, the mi-

[9]Changes in procedure size due to SCBP cause different procedures to overlap and to cross cache

crobenchmarks show large improvements in misprediction rate and show negligible changes in cache traffic. Given these trends in branch predictions and cache misses, we would expect all benchmarks but *go*, *perl*, and *ijpeg* to improve in performance under SCBP.

4.4 Analysis of Performance

Mispredict rate and cache miss rate both influence performance, but the net effect on performance is what matters. To understand the net effect of SCBP on performance, this section examines the magnitude of the branch prediction and cache components and examines their relative importance for different kinds of processors. But because modern processor implementations are complex, we cannot analytically describe all effects of SCBP on performance. To address this complexity, Section 4.5 presents run times for SCBP-transformed programs executing on real hardware.

From our simulations, we know the number of branch mispredictions and cache misses from each program run. The differences between a traditional statically predicted program (history depth of 0) and a program transformed with SCBP (history depth of 15) are shown in Table VII. Negative numbers indicate cases where SCBP improved (reduced) the number of mispredictions or cache misses, while positive numbers indicate cases where SCBP worsened (increased) them. The benchmarks fall into four categories. For the microbenchmarks and for *compress*, *eqntott*, and *m88ksim*, branch mispredictions improve while cache misses change negligibly. Under *li* and *vortex*, both branch mispredictions and cache misses improve. For *espresso*, *gcc*, and *perl*, the improvement in mispredictions is greater than or similar to the worsening in cache misses. And for *go* and *ijpeg*, the worsening in cache misses is much larger than the improvement in mispredictions.

Benchmarks in the first two categories clearly benefit from transformation by SCBP, while benchmarks in the last two categories may benefit from SCBP, depending on the penalties associated with branch mispredictions and cache misses. These penalties vary with processor implementation and cache structure. Table VIII describes some of the penalties and cache sizes associated with recent microprocessor implementations. Multiplying the columns in Table VII by weights for different processor implementations can give some insight into the likely overall change in performance.

By multiplying items from Tables VII and VIII, we can estimate the net performance effect of SCBP. Suppose that a processor has a five cycle mispredict penalty and an average memory access time (AMAT) of seven cycles to serve references that miss in the first-level cache. Then by multiplying the change in mispredictions by the mispredict penalty and adding the product of the change in cache misses with the AMAT, we can estimate the number of cycles that the program will speed up or slow down. For this example processor, we would expect the *go* benchmark from

lines or page boundaries. For example, the original version of *vortex* ended up with its most frequently executed procedure, `Get_MemWord()`, laid out across a page boundary. And the `xlevarg()` routine in the original version of *li* is flushed from the cache when it calls `xleval()`; the SCBP-transformed version of *li* does not suffer from the same flushing behavior. In both cases, a cache-aware procedure-placement algorithm, such as Gloy et al. [1997], would remove this problem.

Table VII. Changes Due to SCBP in Factors that Contribute to Performance. This table shows the difference in counts between the "$k = 0$" (traditional static prediction) and "$k = 15$" (SCBP with a history depth of 15) versions of the program. Simulations were done assuming a 32KB, direct-mapped primary instruction cache.

Benchmark	Change in mispredicts	Change in cache misses
alt	-250,000	5
ph	-299,999	1
corr	-56,667	6
wc	-816,385	0
compress	-3,153,617	17
eqntott	-4,082,420	7
espresso	-2,541,670	505,922
gcc	-651,830	422,969
go	-85,441,578	321,045,879
ijpeg	-29,846,641	50,752,332
li	-167,812,690	-68,303,333
m88ksim	-29,130	2
perl	-19,250,983	19,593,753
vortex	-1,788,406	-112,306,440

Table VIII. Performance-Related Characteristics of Recent Processor Implementations. System board-level caches are reported only for the HP microprocessors, which have no on-chip cache, so the board-level cache serves as a combined primary instruction/data cache in typical workstations.

	Mispredict	Instruction cache			
	penalty	hit time (cycles)		size (bytes)	
Processor	(cycles)	L1	L2	L1	L2
Alpha 21064A	4	2	-	16KB	-
Alpha 21164	5	2	6	8KB	96KB
HP PA-7200	1	1	-	(128KB)	-
HP PA-8000	5	1	-	(256KB)	-
Intel Pentium	2	1	-	16KB	-
Intel Pentium Pro	11-13	1	3	16KB	256KB

Table VII to slow down by $(5 \times -85,441,578) + (7 \times 321,045,879) = 1,820,113,263$ cycles.

4.5 Running Times

Table IX and Figure 31 compare running times of the original programs against programs transformed by overpruning SCBP. In both cases, we ran Pettis-and-Hansen-style basic-block placement after assigning branch predictions. Times were collected using the UNIX `time(1)` command; `time` has a resolution of 0.1 second. All measurements were made on the machine described in Section 4.1. For each benchmark, we profiled and optimized using the training data set and report the time from running on the testing data set. We timed each program 15 times; and we report the arithmetic mean of the last 10 timing measurements. Standard deviations were less than 0.1 for all benchmarks.

The results in Table IX match our expectations from the simulations of performance. *Go* and *ijpeg* slow down, while *compress*, *eqntott*, *gcc*, and *vortex* speed up; the remaining SPEC benchmarks (*espresso*, *li*, and *perl*) show little change. The microbenchmarks perform the same within the resolution of the timer. These results are encouraging: SCBP affects performance as anticipated by our simulations. Mi-

Table IX. Comparison of Running Times for Programs with Per-Branch Static Branch Prediction Followed by Pettis-and-Hansen-Style Basic Block Ordering ($k = 0$) and Programs with Static Correlated Branch Prediction Followed by Pettis-and-Hansen-Style Basic Block Ordering ($k = 15$). We ran each program 15 times and report the average of the last 10 runs.

	Running Time (seconds)			Running Time (seconds)	
Benchmark	k=0	k=15	Benchmark	k=0	k=15
alt	0.1	0.1	gcc	8.0	7.9
ph	0.0	0.0	go	161.4	164.9
corr	0.1	0.1	ijpeg	118.2	120.1
wc	0.2	0.2	li	211.3	211.3
compress	5.2	5.1	m88ksim	2.5	2.5
eqntott	6.9	6.3	perl	60.4	60.5
espresso	3.2	3.2	vortex	40.5	38.7

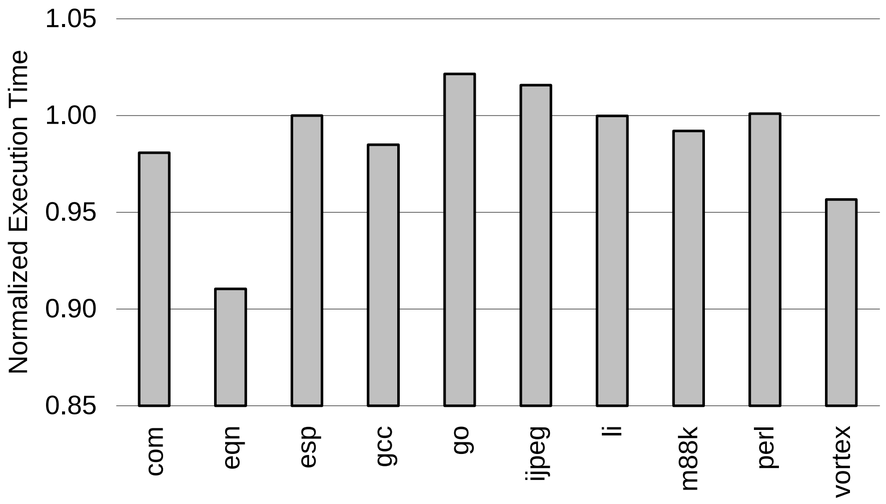


Fig. 31. Normalized change in execution time of the SPEC benchmarks. The original program provides the baseline of 1.0; values below 1.0 represent speedups, while values above 1.0 represent slowdowns (*go* and *ijpeg*). Note that the vertical axis does not begin at zero.

croprocessors with larger cache sizes or more aggressive pipelining will allow SCBP to improve performance overall; these two directions match recent design trends.

To further understand why *espresso*, *li*, and *perl* did not improve in performance, we used the Compaq Continuous Profiling Infrastructure (CCPI) [Anderson et al. 1997] to access the Alpha performance counters and analyze the behavior of these programs. For these three programs, branch predictability improved, but cache misses worsened, swamping the gains from branch predictability. The difference between our simulation results and the timing runs on the Alpha 21164 is the cache size: we simulated a 32KB first-level instruction cache, while the 21164 has only an 8KB first-level instruction cache. Other simulations (not reported here) with smaller cache sizes agree with our timing results.

5. RELATED WORK

Dynamic branch prediction schemes that exploit correlation motivated the development of SCBP and general path profiling; we point to some of the initial work in dynamic correlated branch prediction in Section 5.1. Other researchers have also investigated path profiling, but all such work that we are aware of collects forward, rather than general path profiles. Section 5.2 describes that work. Lastly, Section 5.3 lists other compile-time optimizations with intents or effects similar to SCBP, while Section 5.4 discusses the links between SCBP and DFA minimization.

5.1 Dynamic Correlated Branch Prediction

In 1991, Tse-Yu Yeh and Yale Patt introduced *two-level adaptive* prediction schemes [Yeh and Patt 1991; 1993; Yeh 1993]. These schemes capture *branch history* and use this history as an additional index into the traditional table of two-bit counters. Branch history is recorded in branch history shift registers (BHSRs), where a “1” represents a branch that took (jumped), and a “0” represents a branch that fell through. Two-level schemes may have a single, *global* BHSR or many per-branch or *local* BHSRs (the per-branch BHSR is selected by low-order bits of the branch address). Global history registers capture the pattern of recent branch directions for all branches in the program, while each local history register attempts to capture the pattern of recent directions for a single branch in the program. In their work, Yeh and Patt concatenated a branch history of k bits with the j bits of branch address to index into a table of 2^{j+k} counters. Both the global and local schemes (in Yeh and Patt’s taxonomy, *GAs* and *PAs*) gave better prediction accuracies than per-branch two-bit counter schemes. Figure 32 depicts the structure of a general two-level adaptive prediction scheme. The familiar table of two-bit counters has been extended into a second dimension indexed by the branch history.

Pan et al. [1992] appear to have been the first to use the term “correlation” to describe the behavior exploited by global history schemes. To provide a concrete example of correlation, they showed the code fragment from the SPEC integer benchmark *eqntott* that was reproduced in Figure 1. As noted before, the direction of the third branch is determined if both of the first two branch conditions are true. A global-history shift register with a depth of two or more can exploit this fact, because the shift register records the direction of the previous two branches. When the shift register is used as part of the index into the table of two-bit counters, the deterministic case will map to a different counter from the other three cases, separating out the deterministic case from the other three cases. This separation allows the correlated scheme to achieve higher prediction accuracy than the per-branch scheme; Pan et al. reported up to 11% improvement (reduction) in mispredict rate over per-branch schemes.

While these dynamic branch correlation schemes achieve impressive accuracy, they operate too late to help the compiler. An early motivation for SCBP was to find a way to exploit branch correlation statically, so that compiler optimizations could also benefit from improved prediction accuracy.

5.2 Other Path Profilers

Two other projects of which we are aware also collect intraprocedural path profiles, but in different ways from the path-profiling algorithm described here. Both Ball

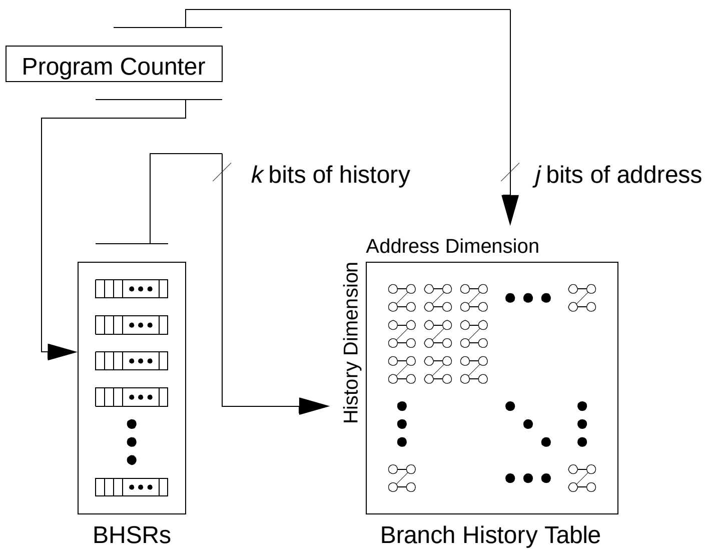


Fig. 32. Two-level adaptive prediction scheme. The branch history shift registers index into a second dimension in the familiar table of two-bit counters.

and Larus [1996] and Bala [1996] profile *forward* paths in the control-flow graph, where a forward path is a path that ends at a back edge or a procedure return. When they encounter a path-ending condition, both other groups record the profiling statistic for the forward path up to that point, then begin recording the next forward path. This is a different method of bounding the number of paths to make descriptions and statistics storage practical.

Forward paths have a very different character from the general, bounded-length paths collected by our algorithm. Forward paths never contain more than one iteration of a loop (although they may capture some of the blocks before a loop entry or after a loop exit). Also, forward paths do not overlap in the trace; instead they "chop" the trace into disjoint pieces. Said another way, our path-profiling algorithm collects statistics over a sliding window of the last k branches in the trace, while the forward path-profiling algorithms collect statistics over disjoint, sequential pieces of the trace.

The algorithm described in Section 2.3 collects execution frequencies. Ammons, et al. [1997] point out that a profile could collect other statistics, such as cache misses or memory-aliasing information, per path using the hardware performance counters of most modern processors. However, using hardware counters with general paths as described in our algorithm can be complex for statistics that do not sum as execution frequencies do. For example, the cache misses along path XYZ are definitely not the sum of the cache misses along paths AXYZ, BXYZ, and CXYZ. Forward paths are much better suited to such statistics, as the hardware counters can be read, then reinitialized between each disjoint program path.

Ball and Larus use an efficient encoding of path descriptions. Before running the program, they compute the number of possible forward paths in the CFG; then they assign each path a unique sequential number. Their assignment of path numbers is carefully chosen so that at each split point in the CFG, a single arithmetic or logical operation suffices to update the path number up to this point. At each back edge and return, the prior sequence of individual operations reproduces the path number by which the program point was reached. Ball and Larus use a low-overhead profiling environment that allows them to scavenge unused registers and spare parallel issue slots on the profiled machine; they achieve impressively low overheads (between 17% and 96%) on SPECint95. Ball and Larus report that they occasionally run out of encoding space for paths in some large procedures; to fix this problem they mark additional CFG edges as places to terminate the existing path and initiate a new path.

Bala's path profiler also scavenges general-purpose registers from the profiled machine, but he uses an encoding of paths that is very similar to the FIFO in our naive path-profiling algorithm. Bala records a unique identifier for the starting branch of a path and the sequence of conditional branch directions that form the path. One limitation of Bala's approach is that he requires special provisions to capture multiway branches. Bala also achieves low overheads (between 10% and 64%) on SPECint95. Bala reports that he rarely encounters sequences of conditional branch directions that overflow the 64-bit registers of modern machines.

Ammons et al. [1997] extended Ball and Larus' algorithm to collect path profiles on the call graph; they call such a profile a *context-sensitive* profile. Because they do not record back edges in the call graph, they will not record recursive call cycles.

5.3 Other Program Transformations

Other researchers have published compile-time optimizations similar to SCBP. Krall [1994] performed a study that examined the prediction accuracy benefits and code size expansion from CFG transformations based on both local and global history. His approach to local history involved building state machines where each state was an entire copy of a loop body, with copies connected based on the local history state of one particular branch in the loop. In this way at most one branch per loop could benefit from improved accuracy without incurring exponential code expansion. Krall very quickly discusses global history (as "correlated branches"); but he is so brief that it is not clear that he built a global reconciler, and it would not be possible from his exposition to build one. Further, Krall's study left cross-validation as further research.

As mentioned in our discussion of global reconciliation, some branches are logically correlated and might be removed by a good optimizer. Mueller and Whalley [1995] combined a path-based examination of the original CFG with partial redundancy elimination (PRE) to find and remove such logically correlated branches. Bodik et al. [1997] present an interprocedural variant of this work. The benefits of this approach are that it does not require any profile information, relying only on aspects of the program code, and that it can remove any correlated branch marked as avoidable. As noted before, SCBP can capture statistical correlation, where a branch direction is made likely, but not determined, by some history. Conversely, SCBP can also be fooled by false correlation: correlation that is specific to one training data set but not general among all runs of a program. Mueller and Whalley's optimization does not capture statistical correlation, but it also will not be led astray by behavior specific to one training set.

Some recent work has applied path profiles to other performance areas, including cache optimizations, memory disambiguation, and instruction scheduling. Gupta, Berson, and Fang used path profiles to drive implementations of PDCE [Gupta et al. 1997] and PRE [Gupta et al. 1998]. Reps et al. [1997] used path profiles in coverage testing. Mowry and Luk [1997] use path profiles to avoid data cache misses; they profile to find paths in the program that are more susceptible to data cache misses than others. Young and Smith [1998] use path profiles during global instruction scheduling; they show that path profiles both simplify the implementation of a superblock scheduler and improve the quality of the scheduled code.

5.4 DFA Minimization

In substantial respects, SCBP is a method to minimize the path CFG without losing any static prediction accuracy. SCBP is thus related to methods that minimize deterministic finite automata (DFA) [Lewis and Papadimitriou 1981]. In DFA minimization, the goal is to eliminate equivalent states: states from which the automaton accepts the same set of strings. Similarly, the goal of global reconciliation is to merge together equivalent sets of paths in the path CFG: sets of paths where the same static predictions will be made in the future. The description of reconciliation given above starts from the original CFG nodes and partitions them until all needed history is preserved. But we could also have built the algorithm to work in reverse, merging equivalent paths to find a minimal partition of the paths leading to a predicted block. We would start with the path CFG (including the likely successor of each path), then merge equivalent paths into sets of paths until we could perform no more merging without affecting the accuracy of likely successor information. Reconciliation differs from DFA minimization because we can merge only paths with the same terminal node, and because the likely successor information can have arbitrary arity (for multiway branches), while acceptance in a DFA has arity 2.

6. CONCLUSIONS

Path profiling provides a new kind of profile information that is intermediate in scope between existing point profiling methods and the collection of full program traces. Path frequencies report how often a sequence of program blocks executed, and they thus precisely indicate the hot regions of the program. Paths are useful for the class of optimizations that attempt to reduce conditional-branch barriers to performance in modern microprocessors.

We examined profiles of general, bounded-length paths in a control-flow graph. We described, implemented, and evaluated a constant-time-per-executed-branch path-profiling algorithm. Using this practical algorithm, we found that the overhead of path profiling is on average within a factor of two of the overhead of edge profiling, and that the resulting database of path profile information is of manageable size for modern processors.

This article described the first optimization to use path profiles: static correlated branch prediction, where we achieve higher static branch prediction accuracy by duplicating and discriminating correlated paths through a program's control flow graph. Our results show that SCBP can improve branch predictability on all of our benchmarks, and that it can improve the performance of the majority of our benchmarks when run on modern machines. The few benchmarks that do not perform well appear to be amenable to more precise overpruning heuristics than the one implemented for this study. With better overpruning heuristics, we can ensure that we never lose in performance when compared to the original program. Furthermore, as future processors implement larger instruction caches and incur larger misprediction penalties, the applicability of SCBP will only increase.

Robustness remains an open issue for path profiles. Initial work in path profiling has shown that different data sets show very similar sets of hot paths [Ammons et al. 1997; Ball and Larus 1996], and this works shows that we can use path profiles in optimization and achieve performance improvements. Even so, in a statistical sense, path profiles must be less robust than point profiles because they derive more statistics (samples) than point profiles from the same trace of program behavior. Since many point-oriented optimizations attempt to generate path frequencies through heuristics (e.g., trace-based schedulers), these optimizations will certainly benefit from path frequency data.

In summary, path statistics can improve performance, but they unfortunately seem harder to apply than traditional profiles. Finding ways to make them work in optimizations requires some facility with graph theory and a willingness to duplicate code so that hot paths can be discriminated from cold ones. As computer architects and compiler writers shoulder more of the burden of improving performance, both will look for new tools and techniques that will improve machine performance. Path statistics and analyses are one such tool; path-driven optimizations like SCBP are some of the techniques.

ACKNOWLEDGMENTS

We thank Josh Fisher, Vas Bala, and Ravi Nair for their strong support of this work. They continually encouraged us and pushed us to improve the algorithms. We also thank the anonymous reviewers for their numerous helpful suggestions.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1988. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass.

Ammons, G., Ball, T., and Larus, J. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.* ACM, New York, 85–96.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. 1997. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comp. Sys. 15,* 4 (Nov.), 357–390.

Bala, V. 1996. Low overhead path profiling. Tech. Rep. HPL-96-87, HP Laboratories. June.

Ball, T. and Larus, J. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE, Los Alamitos, California, 46–57.

Ball, T. and Larus, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Programming Languages and Systems 16*, 1319–60.

Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM, New York.

Bodik, R., Gupta, R., and Soffa, M. 1997. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, 146–158.

Calder, B. and Grunwald, D. 1994. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 242–251.

Fisher, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE TOCS C-30.7,* 7 (July), 478–490.

Fisher, J. and Freudenberger, S. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.

Gloy, N., Blackwell, T., Smith, M. D., and Calder, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, California, 303–313.

Gupta, R., Berson, D., and Fang, J. 1997. Path profile guided partial dead code elimination using predication. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Piscataway, New Jersey, 102–115.

Gupta, R., Berson, D., and Fang, J. 1998. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the IEEE Conference on Computer Languages*. IEEE, Piscataway, New Jersey.

Holloway, G. and Young, C. 1997. The Machine SUIF control flow graph and data flow analysis libraries. In *Proceedings of the Second SUIF Compiler Workshop*. Stanford University, Palo Alto, Calif.

Hwu, W. and Chang, P. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*. IEEE, Washington, D.C., 242–251.

Hwu, W., Mahlke, S., Chen, W., Chang, P., Warter, N., Bringmann, R., Ouellette, R., Hank, R., Kiyohara, T., Haab, G., Holm, J., and Lavery, D. 1993. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing 7,* 1/2 (May), 229–248.

Krall, A. 1994. Improving semi-static branch prediction by code replication. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, New York.

Lewis, H. and Papadimitriou, C. 1981. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, N.J.

Lowney, P., Freudenberger, S., Karzes, T., Lichtenstein, W., Nix, R., O'Donnell, J., and Ruttenberg, J. 1993. The Multiflow Trace scheduling compiler. *The Journal of Supercomputing 7,* 1/2 (May), 51–142.

McFarling, S. 1993. Combining branch predictors. Tech. Rep. WRL Technical Note TN-36, Digital Equipment Corp. Western Research Laboratory, Palo Alto, Calif. June.

Moon, S. and Ebcioglu, K. 1992. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, New York.

Mowry, T. and Luk, C. 1997. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, New York.

Mueller, F. and Whalley, D. 1995. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM, New York.

Pan, S., So, K., and Rahmeh, J. 1992. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.

PETTIS, K. AND HANSEN, R. 1990. Profile-guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.* ACM, New York, 16–27.

REPS, T., BALL, T., DAS, M., AND LARUS, J. 1997. The use of program profiling for software maintenance with applications to the Year 2000 problem. In *Proceedings of the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering.* Springer-Verlag, New York, 432–449.

SMITH, M. 1992. Support for speculation in high-performance processors. Ph.D. thesis, Stanford University, Palo Alto, California.

SMITH, M. 1996. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop.* Stanford University, Stanford, California, 14–25.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM, New York, 196–205.

TORELLAS, J., XIA, C., AND DAIGLE, R. 1995. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture.* IEEE, Los Alamitos, California, 360–369.

WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S., TSENG, C., HALL, M., LAM, M., AND HENNESSY, J. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices 29,* 12 (Dec.), 31–37.

YEH, T. 1993. Two-level adaptive branch prediction and instruction fetch mechanisms for high performance superscalar processors. Tech. Rep. CSE-TR-182-93, Univ. of Michigan, Ann Arbor, MI. Oct.

YEH, T. AND PATT, Y. 1991. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture.* IEEE Computer Society, Los Alamitos, Calif.

YEH, T. AND PATT, Y. 1993. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture.* ACM, New York.

YOUNG, C., GLOY, N., AND SMITH, M. 1995. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* ACM, New York, 276–286.

YOUNG, C., JOHNSON, D., KARGER, D., AND SMITH, M. D. 1997. Near-optimal intraprocedural branch alignment. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.* ACM, New York, 183–193.

YOUNG, C. AND SMITH, M. 1994. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York.

YOUNG, C. AND SMITH, M. 1996. Branch instrumentation in SUIF. In *Proceedings of the First SUIF Compiler Workshop.* Stanford University, Palo Alto, Calif.

YOUNG, C. AND SMITH, M. D. 1998. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE, Los Alamitos, California, 115–123.

Received January 1999; accepted May 1999