Y-Branches: When You Come to a Fork in the Road, Take It[*]

Nicholas Wang Michael Fertig Sanjay Patel
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{*nwang,fertig,sjp*}*@crhc.uiuc.edu*

Abstract

In this paper, we study the effects of manipulating the architected direction of conditional branches. Through the use of statistical sampling, we find that about 40% of all dynamic branches and about 50% of mispredicted branches do not affect correct program behavior when forced down the incorrect path. We call such branches Y-branches.

To further examine this unexpected phenomenon, we provide a characterization of the coding constructs that give rise to such branches. Examples of such coding constructs include short-circuits and ineffectual loop iterations. We provide a statistical breakdown of the frequency of these branches and their constructs. Finally, we suggest some techniques for exploiting this behavior, particularly when it results from short-circuit constructs.

1. Introduction

Control speculation enables high-performance processing for control intensive applications. Ultimately, though, predictability becomes a limiting factor in improving processor performance: at some level, improving the performance of a single thread of execution can be tied to improving the processor's ability to accurately predict and rapidly evaluate branches.

In this paper, we investigate a surprising property of certain dynamic branches. Using fault injection, we find that about 40% of all dynamic conditional branches are *outcome-tolerant*. That is, the behavior of the application is unaffected by whether the particular branch instance is taken or not. Dynamic branches that are mispredicted exhibit a larger degree of outcome-tolerance—about half of all mispredicted branches can proceed down an incorrect architectural path and result in correct execution.

Before we proceed, however, we must carefully define correct execution: an execution path leads to correct execution if it arrives at a correct architectural state (PC, registers, and memory state) without generating software-visible exceptions and before the program performs any external communication. When we say that half of all mispredicted branches are outcome-tolerant, we are stating that such branches can proceed down an incorrect architectural path only to converge on a correct architectural state before the program reaches a system call (which are the only external communications performed by our benchmarks). Often, as we will demonstrate, outcome-tolerant branches reconverge on correct state within a handful of instructions.

In this paper, we perform a characterization of outcome-tolerant branch instances. We call such branches *Y-branches*. We investigate the types of coding constructs that give rise to the outcome-tolerance phenomenon and determine the relative frequency of Y-branches from the various constructs. We also provide characteristics on the architectural state reconvergence latency of a Y-branch.

While the outcome-tolerance phenomenon is far more pervasive than one would at first estimate, it may be difficult to capitalize on this phenomenon. We provide a simulation-based approximation of how exploiting outcome-tolerance might affect a Pentium 4®-like deeply pipelined, dynamically scheduled processor. We sketch some possible architectural extensions that can be used to exploit a subclass of Y-branches associated with short-circuit-like logical expressions.

The remaining sections are organized as follows. Section 2 describes our fault-injection methodology for statistically determining how often the Y-branch phenomenon occurs. Section 3 contains the bulk of our analysis on the various characteristics of Y-branches, including code constructs and reconvergence properties. Section 4 provides discussion on the implication of outcome-tolerance on performance. Related work is presented in Section 5 and our conclusions in Section 6.

[*] Attributed to the baseball great Yogi Berra [2]

2. Methodology

In order to characterize Y-branches, a method for discovering them is necessary. To find all Y-branches for our input sets would require testing all instances of every static branch executed in each benchmark. Since it is computationally expensive to perform this complete analysis, we use statistical sampling. This section describes our experimental setup as well as the applications and infrastructure used to perform the identification process.

2.1. Performing Injection

For each experiment, we randomly sample 1000 conditional branches[1]. To determine whether a branch is outcome-tolerant, we first choose a single conditional branch instance. Then, when the chosen branch is encountered at runtime, the execution of the program is forced down the alternate path. That is, if the conditional branch was resolved to be taken, then the execution of the program is forced down the fall-through path and vice versa. Exactly one injection is made per benchmark execution, so the effects of each of the injections are isolated from one another. Each of these benchmark executions constitute a single trial, and each experiment is composed of 1000 trials for each benchmark. Only conditional branches are targeted in our study—all other control instructions are executed as defined by architectural state.

After each injection is performed, the execution of the program is monitored. We call the execution of the manipulated program the *injected simulation*. To determine the effects of the injection, we compare the injected simulation with a *reference simulation* of the program that is generated without injection. All of architectural state (PC, registers, and memory) is frequently compared against the reference simulation to identify if a *reconvergence point* exists. The discovery of this point guarantees identical execution thereafter as well as confirming the chosen branch is outcome-tolerant.

Most applications do not execute in isolation, so it is important that all external communication is equivalent with that of the reference simulation. The executed applications perform all external communication via system calls. While system calls contain a finite set of inputs, we treat them as barriers for verification. That is, the injected program must reconverge to the original program's architected state prior to the execution of any system call. This restriction reduces *reconvergence likelihood*, however, it also reduces simulation time and complexity.

[1]A random sample of this size yields a confidence interval of approximately $\pm$3.1% with a 95% confidence level. We observed consistent results with repeated experiments of 1000 trials.

Manipulating the control flow occasionally introduces faults that would not have occurred otherwise. Examples include segmentation faults, alignment errors for memory operations, and divide by zero. There are no faults present in the reference executions of each benchmark, so any faults generated in the injected simulations must result from the injections. If we observe any faults in an injected simulation, it is automatically labeled as *non-reconvergent*.

2.2. Simulation and Benchmarks

Our simulation infrastructure is built upon the SimpleScalar 3.0 tool set [3]. More specifically, the instruction-level functional model is the core of our injection and timing simulators. This section describes the process of determining outcome-tolerant branches using the injection simulator. The timing simulation is discussed in Section 4.

The SPECINT 2000 Benchmark suite is used to identify and analyze outcome-tolerant branches. Table 1 provides a list of the benchmarks and their respective characteristics. The dynamic instruction count as well as the static footprint, which represents the number of unique PCs executed, serves as a metric for gauging application size. Given that branch instructions are the focus of this paper, conditional branch instruction statistics are also included to facilitate later discussions. Since system calls are barriers to reconvergence in our experiment, we provide data on these as well. All benchmarks were compiled at optimization level 4 with the Compaq Alpha compiler.

3. Analysis

In this section, we provide a measurement and characterization of the Y-branch phenomenon. Specifically, we find outcome-tolerance rates for various benchmarks, measure state reconvergence latency after an injection, and quantify and characterize the types of coding constructs that lead to outcome-tolerance.

3.1. Outcome Tolerance

Using the methodology described in the previous section, we initially provide a measurement of outcome-tolerance of branches on three bases: by static branch, dynamic branch, and mispredicted branch.

For determining the outcome-tolerance of static branches, we randomly select, before starting a trial, a static conditional branch as the target of the injection. Then, for this static branch, we randomly select a dynamic instance. With this selection method, each static branch is equally likely to be injected.

For dynamic conditional branches, the selection is more straightforward. We randomly select a branch to in-

Benchmark	Inst. Count	Dynamic Branch Count	Dynamic System Calls	Static Footprint (instructions)	Static Branch Footprint (instructions)
bzip2	289M	28.4M	29	8543	840
crafty	625M	27.8M	108	32752	2318
eon	132M	10.4M	186	49268	1776
gap	501M	40.0M	2525	32239	2397
gcc	284M	33.6M	1922	164167	17097
gzip	869M	58.6M	32	9476	828
mcf	411M	53.9M	49	8198	854
parser	513M	71.4M	1070	23402	2390
perlbmk	145M	11.8M	46	34472	2888
twolf	595M	57.7M	403	38463	2896
vortex	272M	24.0M	10562	82037	6297
vpr	534M	46.9M	102	30293	2116

Table 1. Benchmark statistics.

ject out of the set of all dynamic branches. The random selection is done before the trial begins by generating a pseudo-random number between one and the number of dynamic conditional branches in the particular execution of a benchmark.

Outcome-tolerant mispredicted branches are determined by restricting the set of dynamic branches to those that are mispredicted by an 18-bit gshare conditional branch predictor [9]. Again, the random determination is done before the trial begins.

Figure 1 plots the outcome-tolerance rates of the three experiments. Averaged across all benchmarks, approximately 30% of the instances of an individual static branch are outcome-tolerant[2]. Approximately 40% of all dynamic branches are outcome-tolerant, indicating that dynamically frequent branches tend to exhibit more tolerance than infrequent branches. Finally, 50% of dynamically mispredicted branches exhibit outcome-tolerance, indicating that half of all mispredicted conditional branches might be resilient to being mispredicted.

The significant rise in Y-branches in the case of mispredicted branches is particularly pronounced for the benchmarks *bzip2* and *gzip*. The benchmark *bzip2* contains an unrolled loop that accounts for a small portion of the dynamically executed instructions. However, the loop ending branches of the unrolled iterations also account for a large portion of mispredictions. These branches can safely perform early exits since the clean-up loop will perform the remaining loop operations. Similarly, the increase in outcome-tolerance in the benchmark *gzip* is due to three branches that also account for a small portion of the program execution, but represent approximately half of all mispredictions and are highly outcome-tolerant. The loop from *bzip* and one of the three branches from *gzip* are discussed further in Section 3.4.

[2]Because of the statistical nature of our analysis, our averaged results are ±0.9% with a 95% confidence level

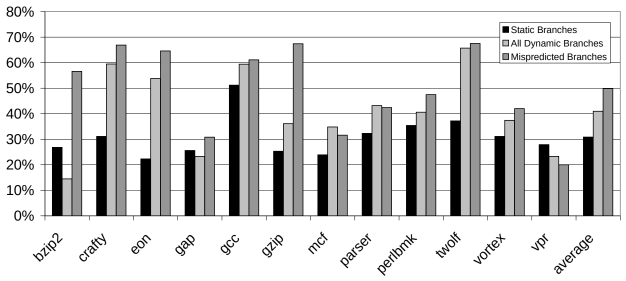


Figure 1. The percentage of branches that are outcome-tolerant Y-branches.

3.2. Reconvergence Latency

In order for a conditional branch instance to be outcome-tolerant, two conditions must be satisfied: (1) the program reaches an architecturally valid program state independent of the branch's outcome, and (2) neither the correct nor incorrect path(s) can see any exceptional situations prior to the reconvergence of state.

This raises an interesting question: how long does it take for a Y-branch to reach equivalent state with that of the reference execution? Figure 2 presents a pictorial representation of the injection of a Y-branch. In this figure, the reference execution represents the stream of instructions executed by the correct architectural execution, whereas the injected execution represents the execution where a particular branch instance is forced down its alternate path. After the point of injection, the two executions execute different instructions. Eventually, if the injected branch is a Y-branch, the two streams converge and execute the same instructions for the remainder of the program—we term this *control reconvergence*. The span in instructions between the point of injection and the control reconvergence point in the injected execution is what we call the control reconvergence latency, which is labeled s in the figure. This same parameter with respect to the reference execution is labeled r. Zero or more instructions later, if the injected branch is a Y-branch, both executions arrive at the same architectural state, or *full state reconvergence*. The full state reconvergence latency is labeled t in the figure.

There is actually another interesting point: *live state reconvergence*, that is some of the full state of the running process is actually dead and therefore not required to be equivalent between the two executions. Live state reconvergence will always occur at or before the two executions reach full state reconvergence. In addition, live state reconvergence is possible in the absence of full state reconvergence. Thus, the set of branches that are outcome-tolerant with respect to live state is a superset of the set of branches that are outcome-tolerant with respect to complete state. Unfortunately, live state reconvergence is difficult and expensive to measure precisely, so we do not investigate it here.

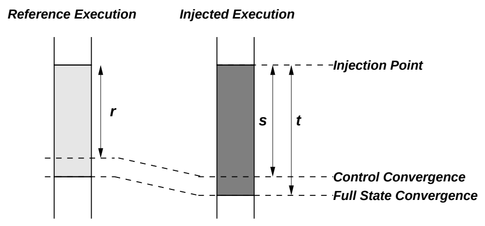


Figure 2. Comparison of reference versus injected executions.

We provide data on the span (in instructions) between the injection of a Y-branch and the subsequent control reconvergence (with respect to the injected simulation), in Figure 3. Because the distributions of the data are very wide, we have only plotted the middle 80% of the distribution, i.e., we exclude the upper and lower 10th percentiles. For example, 80% of Y-branches in *bzip2* tend to control converge between approximately 10 to 100 instructions. The median Y-branch in *bzip2* reconverges on control in 41 instructions, as indicated by the diamond along the vertical line. In our graphs with medians, the average bar represents an averaging across all other percentile bars. With this in mind, the average benchmark reaches control reconvergence in approximately 18 instructions.

Eon differs slightly from the other benchmarks in that many of its Y-branches reconverge on control immediately after the injection. The reason for this is that about a third of the Y-branches are from a single loop-controlling branch. Injecting this branch usually results in an early loop exit which then in turn often results in immediate control reconvergence. Despite the short control reconvergence latency, *eon*'s full state reconvergence latency is relatively slow. We believe this occurs because the early loop exits skip a number of dead stores, where the dead values they would have written have a relatively long lifetime.

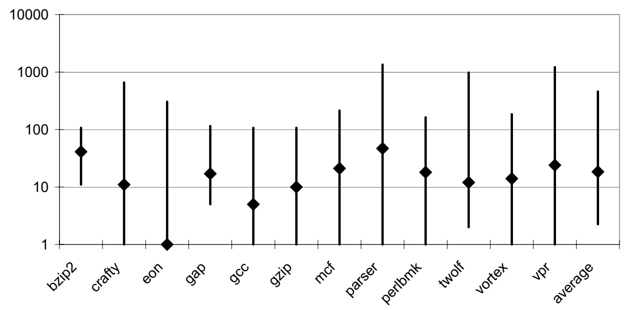


Figure 3. Control reconvergence latency in instructions.

Figure 4 plots the full state reconvergence spans in instructions for the middle 80% of Y-branches for every benchmark. Notice that full state reconvergence spans are much wider ranges than for control reconvergence, ranging from 1 instruction for a Y-branch in *perlbmk* to over 3M instructions on the benchmark *vpr*. Across all benchmarks, the median Y-branch reaches full state reconvergence after approximately 1300 instructions.

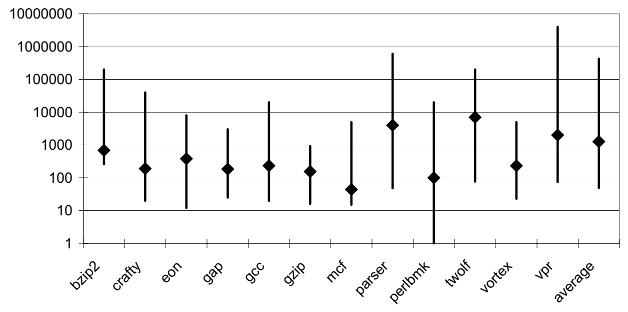


Figure 4. Full state reconvergence latency in instructions.

Figure 5 presents the difference in the number of instructions executed between the two executions, i.e., the value of $r - s$ in Figure 2. Because of the nature of the branch being injected, sometimes the injected execution runs for fewer instructions than the reference execution, sometimes longer (we explain this in Section 3.3). The ranges are provided in this figure (positive numbers indicate that the injected execution ran for fewer instructions). In the median case for the average benchmark, the injected execution ran for nearly the same number of instructions as the reference trace. Somewhat interesting is the fact that in roughly half the cases, the injected execution required fewer instructions than the reference execution to arrive at the same result. This hints at a higher than expected degree of redundancy in dynamic code streams, indicating that additional optimization opportunities exist.

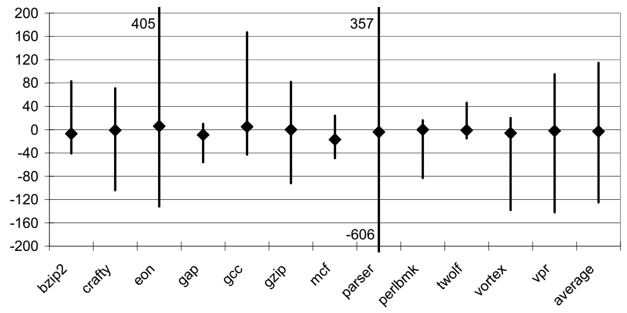


Figure 5. Difference in instructions executed between reference and injected streams.

3.3. Sources of Outcome Tolerance

Outcome-tolerant branches are ultimately due to redundancies introduced by the programmer or compiler. These redundancies occur in various forms, such as superfluous loop iterations, performance-related branches, short-circuits, and the occasional irrelevant decision. Most of these Y-branches are a result of partially dead control, meaning that the branches are outcome-tolerant subject to some but not necessarily all initial states. In this section, we perform a careful dissection of a sampling of outcome-tolerant branches in order to find the underlying reasons behind this phenomena.

What we desire is an ability to examine the control flow graph and possibly the source code of a detected Y-branch in order to characterize the coding constructs that give rise to its outcome-tolerance. To do this automatically, we examine the differences in execution between the reference execution and the injected execution. That is, referring to Figure 2, we examine the differences in PCs executed between the shaded regions of each execution, up until the point of control reconvergence.

Two basic situations arise when we examine the difference between the injected and reference executions, as represented in Figure 6. The leftmost subfigure represents the *Disjoint* case. Here, the injected execution and the reference execution execute completely different static instructions. This situation is primarily due to the Y-branch being an `if-else` construct, where the decision made by the particular dynamic instance of the branch is irrelevant (we provide a specific example of this at the end of this section). Sometimes the reference execution executes more instructions; sometimes the injected execution does.

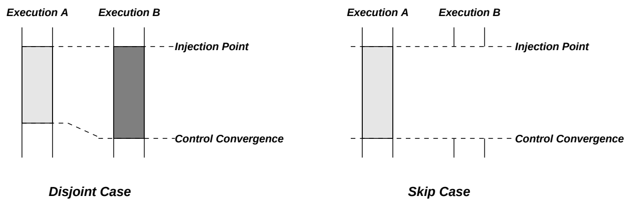
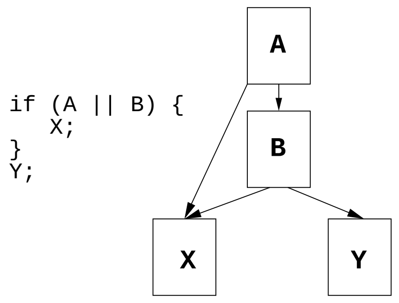


Figure 6. General structure for difference analysis.

The other basic situation is represented in the right subfigure. Here, one of the executions skips over instructions executed by the other starting immediately after the injection point. That is, Execution A might execute blocks X, Y, Z before hitting the control reconvergence point whereas Execution B flows directly to the reconvergence point immediately after injection. For example, if the injection targets a Y-branch that is a loop-ending branch causing the loop to either terminate early or execute additional iterations, then the Y-branch is of the *Skip* category.



```
if (A || B) {
   X;
}
Y;
```

Figure 7. The branch at the end of block A is potentially a Y-branch of the Skip variety.

Skip-type Y-branches can be further classified based on whether loops are involved. If looping is not involved (i.e., none of the branches in the shaded region are backwards branches), the code construct responsible for the outcome-tolerance is most probably of the short-circuit variety. For example, Figure 7 demonstrates the control flow that results from a simple short-circuit expression in C. The branch at the end of block A is an early jump to block X

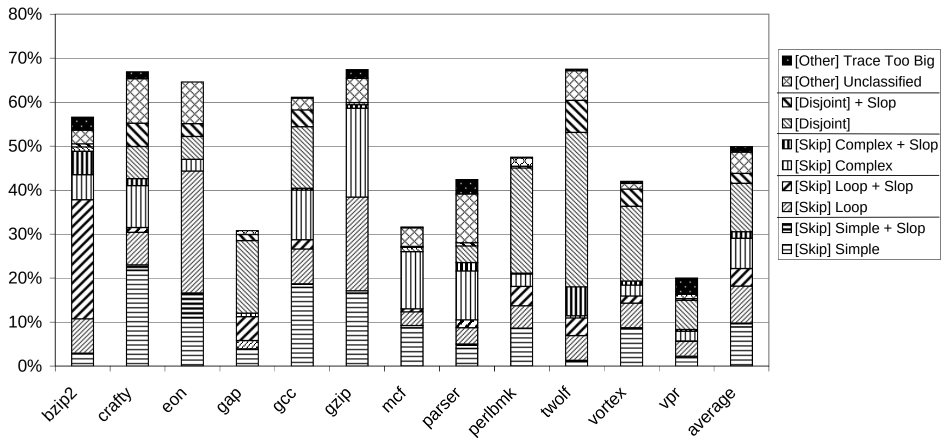


Figure 8. Categorization of Y-branches for mispredicted branches.

if it is determined that A is true. This branch is potentially a Y-branch of the Skip variety when condition B is true and B has no side-effects during evaluation. We call Y-branches of this category Skip/Simple branches. Note that some nested `if` constructs that are not short-circuits also fall into this category.

If the Skip-type Y-branch involves a backwards branch, then we classify the branch as either a Skip/Loop or Skip/Complex branch. Skip/Loop branches are cases where we are confident that the Y-branch is a loop-terminal conditional branch, whereas the Skip/Complex situation is indeterminate: a backwards branch is encountered, so a loop potentially exists in the extra instructions executed. Unfortunately, it is not clear from our analysis whether the Y-branch itself was the loop-terminal branch.

But not all cases are so clean as the categorizations of Skip/Simple, Skip/Loop, Skip/Complex, and Disjoint. Occasionally, the region between the injection and full state reconvergence is too large for our tools to analyze. For cases where this region is over 1M instructions, we categorize it as *Trace Too Big*. Furthermore, cases arise where the Y-branch injection has lingering effects on program state, causing control to randomly waver before finally converging. When this occurs, we sever the traces at the first common instruction and attempt to classify it as described above. These cases are called Skip/Loop+*Slop*, Skip/Simple+*Slop*, etc. In certain cases it is difficult to find a point at which to truncate so we give up (*Unclassified*).

Figure 8 provides a detailed classification of Y-branches that arise from mispredicted branches. From this figure, it is possible to identify the approximate contribution to Y-branches of each coding construct. On average, the contributions to the various coding constructs is about equal based on our categorization. Approximately 10% of mispredicted branches are outcome-tolerant and due to short-circuit like code constructs (Skip/Simple and Skip/Simple+Slop). Approximately 20% of mispredictions are Y-branches that involve loops (Skip/Loop, Skip/Complex, ...). Approximately 15% of them involve `if-else`-type constructs (Skip/Disjoint, Skip/Disjoint+Slop).

3.4. Specific examples

To further illustrate the nature of the outcome-tolerance phenomenon, we examine specific examples, taken from SPEC, of three coding constructs that lead to instances of outcome-tolerance.

3.4.1. Short-circuit example. An example of a short-circuit code construct from *gzip* is shown in Figure 9. The `if` statement is within a `do-while` loop, and it tests four conditions. If any of them evaluate to true, the rest of the current iteration of the loop is skipped by the `continue` statement. In the binary, the compiler implements this `if` statement as four conditional branches, each of which tests one of the conditions and, if true, branches to the bottom of the loop. We observe that the first branch accounts for about 15% of all mispredictions in *gzip* and about 90% of these are outcome-tolerant. In general, the full state and control reconvergence latencies tend to be short for short-circuit like code constructs, and the difference in numbers of instructions executed tends to be small.

3.4.2. Early loop exit examples. *Bzip2* is a file compression program that employs move-to-front coding after applying a transformation on the data to be compressed [4]. When an element is used during encoding or decoding, it is moved to the front of a list, shifting all previous elements down by one entry. The shifting of elements is implemented in a tight loop, which is unrolled four times in the source code. The source code is shown in Figure 10.

```
do {
  ...
  if (match[best_len]   != scan_end  ||
      match[best_len-1] != scan_end1 ||
      *match            != *scan     ||
      *++match          != scan[1])
    continue;
  ...
} while ( ... );
```

Figure 9. Gzip short-circuit example.

The compiler has further unrolled each of these two loops four times, resulting in a 16x unrolled loop, two 4x unrolled loops, and the original loop.

The use of a clean-up loop allows for an early exit of any of the unrolled loops without loss of correctness. Any operations not performed by the unrolled loops will be completed by subsequent clean-up code, at the cost of executing extra iterations of smaller loops. Interestingly, the loop unrolling introduces more mispredictions. For our input set, the number of elements that need to be shifted (and thus the number of original loop iterations required) is highly variable. Unrolling this loop exacerbates this problem by adding more static branches, all of which are highly mispredicted. The branches associated with this source code account for about half of all mispredictions in *bzip2*.

```
tmp == yy[nextSym-1];
j = nextSym - 1;
for (; j > 3; j -= 4) {
  yy[j]   = yy[j-1];
  yy[j-1] = yy[j-2];
  yy[j-2] = yy[j-3];
  yy[j-3] = yy[j-4];
}
for (; j > 0; j--)
  yy[j] = yy[j-1];
yy[0] = tmp;
```

Figure 10. Bzip2 early loop exit example.

Another early loop exit example can be found in *parser* and is shown in Figure 11. This loop is from a function that deallocates a hash by iterating through the hash table looking for hash entries to deallocate. Note that for each loop iteration, if the currently examined hash table entry is `NULL`, then no operation is performed before the loop iterates to the next hash table entry. If the remainder of the hash table entries are empty, then the outside loop (the loop iterating through the hash table) can safely exit early, thus yielding a Y-branch. This branch accounts for approximately 6% of the executed conditional branches in *parser*, and about one-fifth of these are Y-branches.

In general, early loop exit Y-branches yield short state and control reconvergence latencies. This is a result of skipping ineffectual instructions by leaving the loop early. Also, the difference in instructions executed is usually moderate to high, relative to the other main types of Y-branches.

```
for (i=0; i<table_size; i++) {
  for (t = table[i]; t != NULL; t=x) {
    x = t->next;
    xfree(t, sizeof(Table_connector));
  }
}
```

Figure 11. Parser early loop exit example.

3.4.3. If-else example. *Perlbmk* has a function to copy blocks of memory. The function first determines the relative positioning of the source and destination blocks in memory. Based on the relative positioning of these blocks, it executes one of two loops: one copies memory starting from the lowest address to the highest, while the other does the opposite. The function thus guards against the case where the memory blocks overlap. The corresponding source code is shown in Figure 12.

When the segments of memory to be copied do not overlap (which is commonly the case for our input set), the branch that determines which loop is executed is a Y-branch. In addition, the branch is often mispredicted—likely due to lack of a strong pattern in the relative ordering of the memory blocks presented to this function. This branch alone accounts for approximately 16% of all mispredictions in *parser*. On a side note, by identifying this Y-branch, we were able to modify the source code to only execute the poorly predicted branch when necessary. After the modifications were made, most of the mispredictions attributed to this Y-branch were removed. In general, this type of transformation can be used to exploit poorly predicted Y-branches when sufficient conditions for outcome-tolerance can be easily determined.

```
if (from - to >= 0) {
  while (len--)
    *to++ = *from++;
} else {
  to += len;
  from += len;
  while (len--)
    *(--to) = *(--from);
}
```

Figure 12. Perlbmk if-else example.

We have seen that `if-else` type constructs show a wide distribution of reconvergence latencies and instruction count differences. They can vary from short statements to relatively long loops.

4. Exploiting Outcome Tolerance

Despite the surprising frequency of outcome-tolerant dynamic branches, capitalizing on Y-branches for higher performance is not a straightforward endeavor. Partly due to the broad variety of coding constructs that give rise to the phenomenon, and partly due to the trade-offs associated with exploiting them, design and code optimizations that result from this phenomenon are likely to be broad and varied. In this section, we discuss possibilities on leveraging outcome-tolerance to improve program performance and execution efficiency.

Because of the heavy reliance on control speculation (broadly encompassing branch prediction, speculative code motion, and thread-level speculation) for achieving high performance on control-intensive applications, the cost of a misspeculation can be quite high, easily averaging 30-40 cycles on a modern high-frequency pipeline. For thread-parallel processors [10, 12] and other machines that more aggressively evaluate a program's control flow graph, there are likely to be additional costs due to re-execution, re-rename, and re-distribution of computed values. Exploiting outcome-tolerance in some fashion might be an interesting new angle to reduce the overall cost of misspeculation, or to tolerate some misspeculations outright.

Reducing the phenomenon to the three basic code constructs discussed in the previous section (loop exits, if-else branches, and short circuits), we identify ways that specific Y-branches can be exploited.

4.1. Exploiting Loop Exits

Recall the example from Section 3.4 involving the unrolled loop from *bzip2*. Here the loop terminal branches from the unrolled iterations were safely guarded by the clean-up loop (for early loop exits), creating outcome-tolerant branches. In this particular situation, if the cost of mispredicting the loop-ending branch is greater than the sub-par execution resulting from executing the non-unrolled loop, tolerating the misprediction would be worthwhile.

In another vein, some loop-ending Y-branches are weakly specified, meaning that a specific instance of the loop can iterate for a fewer or greater number of iterations without affecting program output. Examples of this include the loop from *eon* that was mentioned in Section 3.2 and the hash deallocating loop in *parser* detailed in Section 3.4. For these types of branches, an injection that causes an early loop exit may result in significantly fewer instructions executed than in the reference execution, indicating the potential for an interesting class of static and dynamic optimizations.

Exploiting outcome-tolerant loop exits in a meaningful manner might be accomplished by the programmer, provided the programmer can be informed of meaningful cases of such Y-branches. Likewise, compiler optimizations might be able to more tightly bound loop iterations and perhaps reformulate the iteration bounds for a loop based on dynamic instance, as accomplished by dynamic code specialization techniques [5]. Threading approaches that use a separate thread or specialized hardware to validate the architectural state of an *approximate* execution thread [13, 14] might also be able to exploit loop-based outcome-tolerance.

4.2. Exploiting If-Else

Nearly a third of outcome-tolerant branches are based on `if-else` constructs (the Disjoint and Disjoint+Slop categories in Figure 8). A particular Y-branch of the `if-else` variety typically performs an unnecessary control between two acceptable options, such as the case highlighted in the memcopy loop for the benchmark *perlbmk* in Figure 12.

As was the case for *perlbmk*, knowing the branch is highly mispredicted and outcome-tolerant enabled a simple programming transformation to remove the mispredictions. The programmer or compiler, once aware of an outcome-tolerant `if-else` branch, can restructure the branch expression to remove the mispredictions.

4.3. Exploiting Short Circuits

One very promising area of exploiting outcome-tolerance deals with Y-branches of the short-circuit variety. These branches arise from the use of short-circuiting operators (i.e. logical AND and OR) in C and C++, and also due to the use of nested-`if` constructs. Due to the semantics of these operators, a compiler will insert early exit branches in the evaluation of such expressions when a term in a logical condition implicates the entire expression, such as the case of `(A || B)` where A is true. The compiler can eliminate this branch in cases where no side effects are possible in evaluating B, but only when any variable references in B are guaranteed to not generate software-visible exceptions.

Predication techniques and conditional move instructions can be used to handle short forms of this control flow but at the cost of *always* fetching and evaluating some unnecessary expressions. What outcome-tolerance indicates is that for such Y-branches, the value of the predicate was not important. In particular, consider the code and pictorial example in Figure 7. If the compiler cannot safely perform B without side effects, then it will insert a short-circuit branch to X after evaluating A. The branch at the end of A is a Y-branch if B is true and evaluating B generates no side-effects.

Based on the data presented in Figure 8, about 10% of mispredicted branches (nearly 20% in benchmarks *crafty*, *gcc*, and *gzip*) are Skip/Simple variety Y-branches, which are likely to be short-circuit style branches.

We suggest an architectural mechanism in which mispredicted short-circuit branches can be tolerated through the use of a *skewed branch*. In a skewed branch, a misprediction does not cause a misspeculation recovery provided that the incorrect branch path ultimately reaches its correct target.

We demonstrate via an example. The essential control flow for a short-circuit style construct is provided in Figure 13. The code construct corresponding to this structure might be, for example, `(A || B || C)`. Suppose branch A is mispredicted to fall through to B despite the expression A being true. If B is true and the control eventually reaches X, then the instance of A is outcome-tolerant. We do not need to act on the misprediction of A because control has correctly converged at the right point of execution.

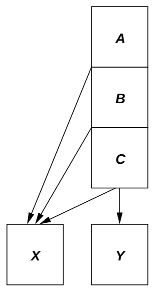


Figure 13. Skewed branches can be used as the internal branches of a short-circuit like region.

For such short-circuit branches, the compiler can use a skewed branch. With skewed branches, a misspeculation is triggered only if the eventual end target of X or Y is incorrect. That is, mispredicting A (or B) to be fall-through is irrelevant provided that control eventually ends up at X. In other words, for conditional structures where `(A || B || C)` is likely to be true, but any one of A or B are hard to predict, the skewed branch approach can mask misprediction penalty. Because of the short-circuiting restrictions, if A is true, no side-effects in B or C should be visible. The use of control speculation-like masking in EPIC [1] and delayed error reporting can be used to mask out side-effects of unintended architectural paths.

It is interesting to note that the difference between using skewed branches in a region such as the one in Figure 13 versus predication is that with skewed branches, not all blocks are fetched and evaluated, which saves on bandwidth when the predictor is able to make a correct taken prediction. Predication on the other hand will always require all conditions be fetched and possibly evaluated.

Lastly we note that compiler transformations (particularly in the compiler we used) attempt to remove short-circuiting branches by establishing that evaluations of subsequent conditions will not generate side-effects (in effect mirroring the predication approach). With the use of skewed branches, the cost of mispredicting a branch is reduced, and it may actually be less costly to reintroduce the branch.

4.4. Performance Approximation

In this section we provide a cursory performance analysis in order to frame the impact on performance of exploiting Y-branches. There are two main methods by which Y-branches can increase program efficiency: by removing the branch misspeculation penalty and by shortening program execution paths. If an outcome-tolerant branch is mispredicted, flushing the pipeline and restarting fetch at the correct address is not necessary, since executing down both paths is equivalent. Also, if the new set of instructions executed can be completed faster than the original “correct” set, the execution time of the program can be reduced via the new execution path. In this section, we use a set of heuristics to identify and target Y-branch mispredictions during a program’s execution in order to estimate their impact on program performance.

It is important to keep in mind that this analysis provides neither an upper nor lower bound to all mechanisms which exploit Y-branches. Rather, it serves as a performance approximation for a simple microarchitectural technique for exploiting Y-branches. The importance of this analysis is two-fold. Firstly, it offers a framing of performance potential with which to compare and contrast other mechanisms. Secondly, techniques to exploit Y-branches may have significant (possibly negative) impact on other microarchitectural components, such as branch predictors. This study offers some insight on the relative impact of these positive and negative factors.

4.4.1. Methodology. There are many mispredicted conditional branches in the execution of a program, and determining whether or not each branch is outcome-tolerant is computationally expensive. So in order to get a reasonable approximation on performance, we employ a simple heuristic filter to find candidate Y-branches. In order for a mispredicted branch to be identified as outcome-tolerant in this experiment, it must (A) achieve full state convergence within 1000 instructions and (B) reduce the number of instructions executed over the correct architectural path.

Furthermore, pertaining to condition (B), we allow the alternative path an additional budget of 40 instructions in order to compensate for the cost of a misprediction recovery. We've observed that 40 instructions is a good estimate of the branch misprediction penalty on our timing model.

It is important to note that even though we do not explicitly select which Y-branch categories to take advantage of, our identification and selection mechanisms are biased towards certain types. For example, Skip/Simple Y-branches usually exhibit shorter reconvergence latencies, which fall before our 1000 instruction reconvergence limit. On the other hand, other types of Y-branches such as Skip/Loop may have their reconvergence point after our limit, which subjects them to filtering.

After a decision has been made to follow a non-architected path, it is possible to encounter another mispredicted branch prior to reaching the reconvergence point. If this occurs, the newly mispredicted branch can be analyzed for outcome-tolerance in the same way as any other branch. This is true because the new branch, although not in the original instruction stream, can be treated as such since it has already been identified as part of an equivalent path of execution.

This process of finding and selectively choosing Y-branches is repeated until the end of the program is reached. At this point, a new and valid path has been defined through the program, and this path can be followed by a timing model to simulate how a real microprocessor would react. The timing model uses the branch predictions laid out by the functional simulator and does not recover from mispredicted branches when the predicted direction is along the predefined (and guaranteed to be equivalent) path. The model simulates a modern 8-wide dynamically scheduled 15-stage pipeline microprocessor with significant branch misprediction penalties (due to the deep pipeline).

4.4.2. Results. Figure 14 shows the percentage of mispredicted branches that we capture in our experiment. On average, 26% of mispredictions in the new path are Y-branches and have fewer instructions along the predicted path. Unfortunately, sometimes mispredictions are introduced into the new path by not recovering from previous mispredictions. For example, in *gcc* the number of mispredictions encountered along the new path is more than doubled. Even after identifying 34% of these as Y-branches (and thus removing the associated misprediction penalty), there are still more mispredictions than in the original execution.

Figure 15 shows the speedup obtained. On average, 6% speedup is seen over the baseline execution that has no manipulated Y-branches. *Gzip* sees 38% speedup, which is a culmination of a few main factors: fewer mispredictions along the new path, the removal of 40% of these fewer mispredictions, and a reduction in instruction count by 5%. On the other hand, *gcc* suffers a -6% speedup, mainly due to the effect discussed in the previous paragraph. These results indicate that our heuristic for choosing whether to recover from a branch misprediction or to simply follow the predicted path requires retooling—there are many more factors in play than instruction count differences. However, *gzip* and *mcf* show promise that significant speedups can be obtained from taking advantage of the Y-branch phenomenon, despite all of the restrictions we have imposed.

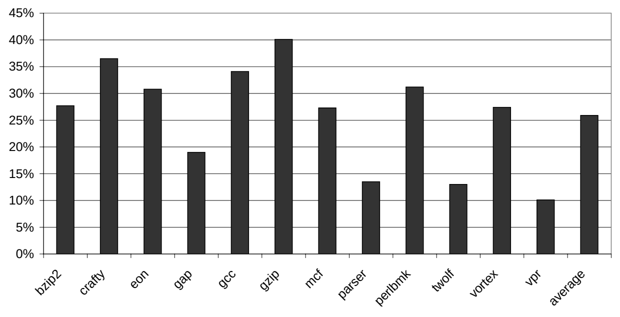


Figure 14. Percentage of mispredictions removed along the new execution path.

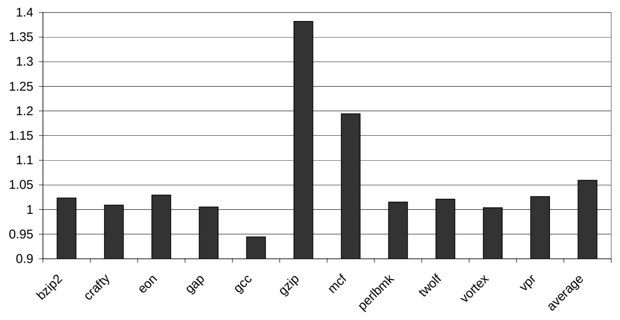


Figure 15. Speedup obtained along new path.

5. Related Work

This section covers the set of work related to outcome-tolerant branches. The related work falls into the following categories: (i) analysis of fault injection on the control path of the program, (ii) studies on redundant code, and (iii) the use of control flow information to detect faults.

Czeck and Siewiorek [6] performed fault injections at the gate-level to provide a basis for fault models at the application level. During their analysis, they described scenarios where control flow could diverge to incorrect, yet valid instruction addresses. However, they did not explore the effects of these, possibly outcome-tolerant, branches.

Gu et al. [8] analyzed the effect of introducing faults into the Linux kernel to understand its response to transient errors. Three fault injection campaigns were run that targeted specific components of the kernel. One of these campaigns, reversing the direction of conditional branches, specifically targets potential outcome-tolerant branches. The authors found that 33% of these types of injections did not manifest as errors. An example of source-level redundancy is given that offers an explanation for this phenomenon, however, further characterization is not done.

Rotenberg [11] explored the effects of removing ineffectual code from a program's execution. Ineffectual code is defined as dead or silent instructions as well as correctly predicted branches. The paper focuses on transformations to the original, architected path that do not alter the program's original traversal of the control flow graph. Our work differs in that we focus on modifying the original path to obtain an alternate, yet correct path of execution.

There has been extensive research on detecting faults using control flow paths. More specifically, Eifert and Shen [7] proposed a method for identifying faults by encoding the instruction bit patterns for instruction regions ending in multiple target branches. A watchdog type processor utilizes these encodings to detect runtime faults.

6. Conclusion

This paper explored the concept of outcome-tolerant branches: conditional branches which, irrespective of outcome, still enable correct execution. To better understand this phenomenon, an investigation into the coding constructs that gave rise to these effects was performed. We found that constructs such as short-circuit like logical expressions and ineffectual loop iterations were responsible.

An approximation of the benefits obtained on a deeply pipelined processor was performed. Furthermore, possible extensions to predication were proposed to harness the outcome-tolerant branches that fall into the short-circuit like logical expression category.

7 Acknowledgments

We thank the other members of the Advanced Computing Systems group as well as Steve Lumetta, Scott Mahlke, and the anonymous referees for providing feedback during various stages of this work. This material is based upon work supported by the National Science Foundation under Grant No. 0092740 and the C2S2 Marco Center, with support from AMD, Intel, and Sun Microsystems.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the impact epic architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[2] Y. Berra and D. Kaplan. *When you Come to a Fork in the Road, Take It! Inspiration and Wisdom from One of Baseball's Greatest Heroes*. Hyperion, 2001.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] C. Consel and F. Noel. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN/SIGACT Symposium on the Principles of Programming Languages*, 1996.

[6] E. W. Czeck and D. P. Siewiorek. Effects of transient gate-level faults on program behavior. In *20th International Symposium on Fault Tolerant Computing, Digest of Papers*, pages 236–243, June 1990.

[7] J. B. Eifert and J. P. Shen. Processor monitoring using asynchronous signatured instruction streams. In *14th International Symposium on Fault Tolerant Computing, Digest of Papers*, pages 394–399, 1984.

[8] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *International Conference on Dependable Systems and Networks*, 2003.

[9] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[10] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[11] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Technical report, North Carolina State University, November 1999.

[12] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1992.

[13] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[14] C. Zilles and G. Sohi. Master/slave speculative parallelism. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.