HydraScalar: A Multipath-Capable Simulator[*]

Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
skadron@cs.virginia.edu

Pritpal S. Ahuja
Alpha Advanced Development
Compaq Computer Corporation
Shrewsbury, MA 01545
pritpal.ahuja@compaq.com

Abstract

Even sophisticated branch-prediction techniques necessarily suffer some mispredictions, and even relatively small mispredict rates hurt performance substantially in current-generation processors. This suggests the study of multipath execution, *in which the processor simultaneously executes code from both the taken and not-taken outcomes of a branch.*

This paper describes HydraScalar, a simulator built on the SimpleScalar toolkit (version 2.0) to accurately model a wide-issue, out-of-order execution multipath processor. The paper describes the simulator's mechanisms for instruction fetch, branch handling, branch-confidence prediction, and multiple-path management. Because multipath execution is so sensitive to branch prediction, HydraScalar significantly expands SimpleScalar's modeling of branch and instruction-fetch effects, therefore adding a number of features that are beneficial regardless of whether multipath or conventional superscalar, out-of-order execution is being modeled.

1 Introduction

1.1 The Need for Multipath Execution

Modern processors employ a variety of sophisticated branch-prediction schemes to avoid delay penalties imposed by conditional-branch resolution. Correct predictions of branch outcomes can almost eliminate these penalties. But mispredictions that remain still cause serious disruptions in instruction flow, as the processor wastes time following the wrong path until the misprediction is detected. As issue widths increase and processor pipelines deepen, the misprediction penalty increases.

Despite many advances in branch prediction, many programs still suffer a substantial number of mispredictions. Since the delays caused by conditional-branch mispredictions remain a serious problem, we [1] and other groups [7, 12, 13, 31, 32, 33] have investigated a different kind of remedy: the simultaneous execution of *both* the taken and not-taken instruction sequences following a conditional branch, with cancellation of the one that turns out to be incorrect when the branch is finally resolved. Because additional branches are likely to be encountered before the first branch is resolved, most of these research efforts consider the possibilities and potential benefit of executing more than two paths simultaneously. Each time *multipath execution* forks successfully, it eliminates a misprediction, although possibly with the expense of increased hardware contention.

Ideally, forking would happen only on mispredicted branches. A *confidence predictor* [9] attempts to evaluate the likelihood that a branch has been correctly predicted. Multipath execution uses confidence prediction to reduce hardware contention: instead of forking, the processor speculates conventionally past higher-confidence branches, following only the predicted path.

[*]This work was done while both Skadron and Ahuja were in the Department of Computer Science at Princeton University.

Multipath execution requires additional hardware (and simulation detail) throughout the processor, of course. The most affected section is the instruction fetch unit, which needs extra resources for each path. Since more instructions will be in flight at once, additional functional units, registers, cache ports, and so on are also needed, but the structure and control of this part of the machine (and simulator) are very much like contemporary multi-issue dynamic-execution processors.

Our experiments [1] showed that multipath execution can offer sizeable IPC improvements over more traditional single-path execution models: 4 paths yield speedups of up to 30% on future-generation processor configurations. For programs with very high accuracies which are unlikely to benefit from multipath execution, our experiments show that performance does not decrease. Although 4-path speculation would have a significant hardware cost, there are extensive overlaps with the hardware required by multithreading approaches such as simultaneous multithreading (SMT) [16], and combinations of multipath and multithreading make efficient use of such hardware [33].

We also demonstrated that total instruction fetch bandwidth is a key lever on performance, and that heuristics to devote extra fetch resources to probably-correct execution paths can help further improve performance. Klauser and Grunwald [11] and Santos *et al.* [21] have further explored how to tune a multipath processor to gain additional performance increases.

1.2 HydraScalar

In this paper we describe HydraScalar, the simulator we used for our multipath studies and subsequent work on branch prediction. It is derived from the Wisconsin SimpleScalar 2.0 toolkit's *sim-outorder* simulator [2]. HydraScalar models multipath execution, including confidence prediction, return-address-stack management, fetch from multiple paths, register renaming, and the intermingling in the processor core of instructions from different paths. It also includes features to model speculative branch-history update with fixup [24] and speculative return-address-stack update with fixup [22], as well as additional features for modeling today's aggressive pipeline organizations.

In addition to the core simulator, the HydraScalar distribution augments the SimpleScalar toolkit in several ways, like the addition of new branch-predictor models.

HydraScalar is made publicly available under the same terms as the SimpleScalar toolkit. It is therefore free for non-commercial users and is subject only to the restrictions that the copyright notice must accompany all re-releases of HydraScalar and that third parties are forbidden to place any additional distribution restrictions on extensions to HydraScalar. It is distributed "as is" in the hope that it will be helpful to the research community. It comes with no warranty, and no author or distributor accepts any responsibility for the consequences of its use. It can be obtained by contacting the authors. Note that a copy of SimpleScalar 2.0 is also required for use of HydraScalar.

The remainder of this paper is structured as follows. The next section describes the basic structure of HydraScalar and gives an overview of its new features. Section 3 then describes in detail the additional support required to model multipath execution, and Section 4 describes new capabilities for modeling branch prediction and branch-history update timing. Finally, Section 5 concludes the paper.

2 HydraScalar Mechanics

2.1 SimpleScalar Toolkit

HydraScalar is built on the SimpleScalar 2.0 toolkit, which provides a set of modules for building processor simulators, as well as several sample simulators. SimpleScalar-based simulators interpret executables compiled by *gcc* version 2.6.3 for a virtual instruction set closely resembling MIPS IV [19] called the "portable instruction set" or PISA. These simulators instantiate a virtual machine and emulate the object program's execution in order to accurately simulate behavior on mis-speculated paths.

Its speed, flexibility, and portability mean that SimpleScalar now enjoys widespread use for both research and teaching purposes. The toolkit's *sim-outorder* simulator in particular has received widespread use in the research literature. It models a wide-issue, out-of-order processor. Unfortunately, it is not well suited to work on multipath execution, because it cannot simultaneously manage multiple paths or track rollback of mispredicted branches that already lie on a mis-speculated path. (State is only maintained for a branch that beings mis-speculated execution, so rollback due to subsequent mispredictions on that mis-speculated path is lost.) HydraScalar therefore uses the pipeline model from sim-outorder but re-implements the way sim-outorder tracks state, and HydraScalar also adds a number of new features, especially in the area of branch prediction.

2.2 HydraScalar Extensions

Like a real processor (and sim-outorder), HydraScalar conceptually checkpoints necessary state as it encounters branches and then proceeds down the predicted path or down both paths, executing wrong-path instructions as appropriate. In fact, SimpleScalar and HydraScalar emulate instruction outcomes early in the pipeline so that they can detect early which branches are mispredicted and simplify later pipeline modeling. This means they need only checkpoint state for mispredicted branches (or, with multipath execution, any forked branch). Unlike sim-outorder, HydraScalar checkpoints state for all mispredicted branches, even if they already lie on a mis-speculated path. These later mispredicted branches may actually resolve before their predecessors, so HydraScalar can model the effects of their rollback. The checkpointing is actually performed by detecting mis-speculated execution and having all updates to architected state write to scratch state that can simply be discarded once the mis-speculation is discovered. To track multiple levels of mis-speculation, HydraScalar simply keeps multiple levels of scratch state.

Upon detecting a mispredicted branch or a forked branch, wrong-path instructions are squashed, and recovery from the checkpointed state is straightforward. Instruction fetch is redirected; the appropriate paths are terminated; the appropriate number of levels of scratch state are discarded; and in-flight instructions from terminated paths are squashed. Because instructions from multiple paths are interleaved in the instruction window, squashed instructions leave holes that are eventually freed at commit. This is accomplished by broadcasting the path ID of the misspeculated path. The construction of the path IDs results in that path and all child paths' being terminated. This is described in further detail in the next section along with other aspects of modeling multipath execution. The rest of this section describes the core HydraScalar model.

HydraScalar Features. In addition to modeling multipath execution, HydraScalar extends sim-outorder in the following ways:

- Modeling the effects of multiple levels of mis-speculation. (Described above)
- Modeling different options for branch-history update timing, as described in [24]. (Section 4.2)
- Modeling different options for return-address-stack update, as described in [22]. When modeling multipath processors, private copies of the return address stack are given to each path. (Section 4.1)
- Modeling a cap on the number of in-flight branches to account for finite shadow state. (Section 4.3)
- Modeling a set-associative branch history table (BHT). (Section 4.3)
- Modeling alloyed branch prediction, as described in [26]. (Section 4.3)
- Modeling a more flexible hybrid branch predictor, as described in [3]. (Section 4.3)
- More accurate modeling of the interaction between the instruction-fetch engine and the instruction cache. (See below)
- More detailed modeling of an out-of-order processor's pipeline, including extra pipeline stages and separate integer and floating-point issue constraints. (See below)
- Modeling a finite number of MSHRs. (See below)
- Support for reduced simulation times. (See below)

Instruction Fetch. In modeling the interaction between instruction-fetch and the I-cache, HydraScalar takes account of the fact that in superscalar processors, the unit of instruction fetch is larger than a single instruction (*e.g.*, 4 instructions). This gives more realistic I-cache miss data. HydraScalar also improves fetch modeling by recognizing that any particular fetch block should come from a single cache line. Of course, to facilitate wide-issue multipath processing, different paths can fetch from different cache lines, and individual paths can fetch multiple contiguous blocks. In other words, a particular path can predict multiple branches per cycle and fetch past not-taken branches, but cannot fetch past taken branches.

The confidence predictor responds in a single cycle. This is plausible because this matches the response time of the branch predictor, and our confidence predictors use simpler hardware. Instruction fetch down the forked path's speculative path can commence on the cycle after the fork decision.

Pipeline and Issue. HydraScalar also allows modeling of variable pipeline lengths. Under the assumption that the simplest operations still take a single cycle to execute, the additional stages are added after decode. These allow for the modeling of additional decoding, renaming, enqueuing, and so forth. Additional stages for fetch (due to a multi-cycle I-cache) can be added by modeling the longer cache latency and the corresponding increase in the branch misprediction penalty. The HydraScalar pipeline therefore looks like

```
IF ... ID ... EX WB CT
```

with branches resolved at the end of the EX stage and recovery taking place during WB.[1] Note that an extra stage for register read is not modeled, something that would further boost the model's flexibility.

[1]IF = fetch, ID = decode, EX = execute, WB = writeback, CT = commit.

Renaming looks in a register mapping table to determine whether operands reside in the RUU or have been committed to architectural state. This mapping table is copied when a new path is forked, and a shadow copy of the table must be saved each time the processor speculates past a branch. If all a path's shadow maps are full, fetch stalls for that path. As mentioned, the number of in-flight branches (and hence the amount of number of shadow-state entries that are modeled) is a parameter.

HydraScalar also allows modeling of different instruction-window and issue topologies. HydraScalar can model either a single, unified, reorder-buffer and issue-window for all instruction types (a *register update unit* or RUU [28]), which is the basic sim-outorder model and resembles the HP PA-8000 [8] for example; or a separate reorder buffer with smaller integer and floating-point issue queues, which resembles the Alpha 21264 [5] and the Intel Pentium-Pro/II/III [4] for example. Regardless of the model, issue selects the oldest ready instructions, independent of path.

Cache. HydraScalar uses SimpleScalar's cache model but extends its model of non-blocking loads by adding MSHRs [14] to model a finite number of in-flight loads. The bus model has also been extended to model a split transaction bus. Different levels of cache can have different bus latencies. The latency accounting has also been extended to separate the time to probe the cache for hit/miss from the time for data transfer. Frontside/backside off-chip caches can be modeled by setting this parameter appropriately.

Simulation Support. Because cycle-level simulations are slow, and HydraScalar is even slower than sim-outorder, benchmarks cannot typically be run to completion. This necessitates some method of reducing execution time. Use of small inputs, like the SPEC [29] "test" or "train" inputs, is controversial, so we have followed the practice of identifying a representative segment of the program's execution and simulating that. This approach is described in [23]. The most important aspect of this approach is to fast-forward past any unrepresentative behavior while the program starts up, because many benchmarks exhibit startup phases. Otherwise this unrepresentative startup behavior will be disproportionally represented in the sampled simulation. A number of benchmarks exhibit such unrepresentative behavior; SPECint95's *compress* and *perl* are particularly stark examples of this. Once simulation has been fast-forwarded to the desired portion of the program, a segment of any desired length (*e.g.,* 50 million instructions) can be simulated in full detail and then the simulation can be terminated. During fast-forwarding, cache and branch-predictor state are updated to avoid unrealistic cold-start effects in these large structures. After switching to full-detail mode, the simulator structures can be primed for a further period of time (to warm up the instruction window, etc.) before statistics-gathering begins.

3 Modeling Multipath Execution

At those conditional branches where it is difficult to predict the correct execution path, simultaneously executing both paths prevents losing cycles to misprediction and recovery. Each resulting path may reach further branches; given sufficient hardware, control may fork once again. Branches which do not or cannot fork are predicted and speculated conventionally. Once a branch resolves, the wrong path and any of its child paths are squashed. The benefits from eliminating mispredictions in this way are tempered by the increased contention for resources as paths divide.

3.1 Differences from Conventional CPUs

The baseline hardware modeled by HydraScalar has three main differences from current high-performance CPUs:

Confidence Prediction and Fork Control Unit. At each conditional branch, deciding whether to fork an additional path depends on *(i)* whether there are any path resources currently free and *(ii)* whether the branch is likely to be mispredicted. Item *(i)* is easy to track, but item *(ii)* is more difficult. A simple policy would be to spawn a new path at every conditional branch as long as resources are available; but we found that this *naive* forking performs poorly because it consumes the available path contexts with unnecessary forks. The upper bound on the performance of all such schemes is demonstrated by a strategy that forks precisely at the right time, on every branch misprediction (when path resources are available). We refer to this as *omniscient* forking. Note that this can still perform worse than omniscient branch *prediction*: forking can still create resource contention, and if mispredictions are tightly clustered, forking contexts may be exhausted. More realistic and elaborate policies try to reduce hardware requirements by being more precise in determining when the branch will be mispredicted. For these policies, we model a dynamic confidence-prediction unit that operates in parallel with the branch-prediction unit.

HydraScalar can model forking that is based on several different types of hardware-based confidence predictors as well as naive and omniscient forking. While the branch predictor returns its prediction of whether or not the branch will be taken, the confidence-predictor returns its prediction of whether or not the branch predictor will be correct. If a context is available and a fork is indicated, HydraScalar performs the fork, initializing a new context and assigning new path IDs to *both* the taken and not-taken paths (see below for more information about path IDs).

Confidence Prediction Options. Confidence-based forking mechanisms can be constructed from two orthogonal components: a policy for counting mispredictions of recent branches, and a policy for applying this counting history.

Ways to count mispredictions were first discussed by Jacobsen, Rotenberg, and Smith in [9]. HydraScalar supports combinations of the following, with configurable table sizes and thresholds. Overall, we found that profiling approaches worked almost as well as the basic mechanisms from [9].

Hardware State:

No Hardware Table. If no hardware state is kept, then we either revert to the naive policy discussed earlier or use a static, profile-based scheme discussed below.

Ones Counter. The simple hardware-based confidence predictors considered here use a table indexed by the branch PC. With a *ones-counting* scheme, each entry in the table is an n-bit shift register. These bits represent the accuracy of the branch predictor for the last n branches that mapped to this entry. If the number of "corrects" exceeds a certain threshold, the confidence predictor predicts high confidence.

Saturating Counter. Each time a branch is correctly predicted or mispredicted, the entry's contents are incremented or decremented. A value greater than a specified threshold indicates high confidence.

Resetting Counter. As with saturating counters, correctly-predicted branches cause the contents of the appropriate table entry to be incremented, but when a branch is mispredicted, the entry's contents are reset to 0, not merely decremented.

Means for Applying Confidence State:

We can extend the basic techniques above by incorporating policies for *applying* knowledge from the hardware state.

No Policy. This reverts to one of the basic techniques above.

Profile-Based. Any of the above schemes can be applied in a profile-based manner. In profile-based approaches, each conditional branch is classified based on observed misprediction statistics from a pre-run profiling step. Profile-based confidence can be performed without a hardware table [9]—branches are statically placed in either the "do not fork" or "fork aggressively" categories. The processor attempts to fork only when a branch from the latter category is encountered. We found that assigning all branches with a greater than 35% misprediction rate to the "fork aggressively" category is successful. Other cutoffs (*e.g.,* 40%, 50%, 70%, etc.) were also tried, with lesser success, although that may change for non-SPEC95 programs.

When combining profiling with dynamic confidence history, each category of profiled branches has a different forking threshold. The count for a branch's table entry is compared to the threshold for its class; if the former exceeds the latter, the CPU does not fork.

Resource-Based. With a finite number of path contexts, the CPU cannot always fork when it wants to. If the CPU has reached its forking capacity, subsequently-fetched branches cannot fork until some contexts are freed. Therefore the CPU should sometimes refrain from forking on a branch that is marginally low-confidence, in case an even lower-confidence branch soon follows. Resource-based schemes address this. These schemes use the same counting tables discussed above, but with multiple thresholds to determine whether to fork. When fewer contexts remain free for forking, these schemes use stricter (*i.e.,* lower) forking thresholds. This makes it easier for a branch to be rated high confidence, reducing the likelihood that the CPU will be unable to fork on a truly low-confidence branch later on.

Combining Profile-Based and Resource-Based Approaches. Incorporating hardware availability (*i.e.,* number of forking contexts available) with a profile-based predictor is also possible. Profiling now categorizes branches into, categories, say 0, 1, 2, and 3. The lower the category, the lower the confidence (and the more aggressive the attempt at forking). When deciding whether to fork, a branch's category is now compared with the number of available forking contexts. If the branch's category is less than the number of free contexts, then fork. Thus, category-0 branches are always forked if any free contexts are available. Category-*N* branches are never forked given a machine with *N* contexts.

Multiple Path Contexts. Multipath execution assumes multiple path contexts to allow several control flows to execute concurrently. In particular, the fetch unit must fetch from multiple paths each cycle.

The essential hardware for each path is:

- a program counter
- a copy of, or its own port into, elements of the instruction-fetch unit, including the instruction cache, branch predictor, and confidence predictor
- a separate return-address stack—*not* a port into a unified one (Section 4.1)
- a register map, and shadow register maps sufficient for any unforked (conventionally speculated) branches which execution might have to unroll

The fetch unit tags instructions with a path ID. Since each path has its own register map, renaming remains essentially unchanged from a conventional CPU; there are no renaming dependencies among paths. Predecode bits can be maintained in the instruction cache, indicating dependencies among instructions fetched for a particular path. This further speeds up renaming and ensures that all the instructions fetched in a cycle can be renamed quickly.

The instruction window contains a mixture of instructions from active paths. Instructions are tagged with a path identifier. One could instead provide separate instruction windows per path, but that approach would give poor performance when only a small number of paths are active, *i.e.,* when the program has few mispredicted branches.

Issue remains unchanged. Renaming ensures a coherent view of the register space, so instructions may arbitrate for execution as soon as their operands become ready. Context tags in the load-store queue ensure that paths see data flow through memory from only the appropriate path.

Path IDs and Selective Misprediction Recovery. In a single-path processor, the CPU handles a mis-speculated path by squashing all the instructions in the RUU after the mispredicted branch. In a multipath processor, however, instructions from many paths—including the correct path—may exist simultaneously and interleaved in the RUU. The CPU must have the ability to selectively squash only those instructions that belong to a particular mis-speculated path and its children. To accomplish this, every path has a unique *path ID* which encodes the forking history of recent branches using binary-prefix notation. New IDs are created when a branch forks, and IDs can be recycled. If a forking branch's ID is x, then the taken path's ID is $x1$, and the not-taken path's ID is $x0$. When that branch resolves, it broadcasts the correct path's ID, say $x1$. The CPU then squashes all instructions that follow the branch and are on the wrong path or a descendent, *e.g.* on paths $x0$, $x00$, $x01$; instructions from $x1$, $x10$, $x11$, etc. are spared. A similar idea was independently proposed by Klauser, Paithankar, and Grunwald [12] and Kol and Ginosaur [13]. Non-forking, conventionally-speculated branches broadcast similarly, but for a mispredicted branch no instructions from the correct path exist. Squashed instructions turn themselves into NOPs. The resulting "holes" propagate like normal instructions but are ignored until commit, when they are reclaimed.

The path IDs are implemented as circular bitmaps, with a global head pointer and a per-instruction tail pointer. The global head pointer indicates the oldest active forked branch. An instruction's tail pointer indicates how many subsequent branches have forked. Together, the pointers indicate what portion of any bitmap contains useful information: newer instructions, for example, have a tail pointer farther from the global head pointer, because more intervening branches have been seen. Since branches may resolve out of order, and the global head pointer cannot advance until the oldest forked branch retires, bitmaps could grow until the tail pointer would overtake the head. Forking halts at this point. But the bitmap length is equal to the depth of the forking tree: unless the tree is long and narrow and some branch takes an unusually long time to retire, bitmap length is not a problem. A bitmap needs to be at least as long as the log_2 of the maximum number of outstanding paths.

A per-branch pointer must be stored as well, indicating which bitmap position corresponds to each branch. When a branch retires and its path's local head pointer matches the global head pointer, the global head pointer can be advanced.

3.2 Similarities to Conventional CPUs

It is important to note that the back end of a multipath CPU is essentially a conventional CPU augmented with extra issue and execution capacity to handle the extra running paths, and HydraScalar can therefore use most of sim-outorder's pipeline model and all of its functional simulation.

3.3 Fetch Bandwidth

To avoid wasting fetch resources along incorrect paths, HydraScalar also supports several heuristics for priority-based allocation of fetch bandwidth among the executing paths. While we cannot know the correct path *a priori*, we can note that the predicted path—the path indicated by the branch predictor—is correct most of the time. Since the paths other than the predicted one are often wrong, performance often improves when the predicted path gets more I-fetch bandwidth than other paths. Applied too extremely, however, paths starve and behavior reverts to the single-path case.

Four schemes are supported: a simple round-robin policy ("simple"), a round-robin policy that ensures the predicted path can fetch every cycle ("pred-pri"), a predicted-path-priority policy that allows non-predicted paths to fetch only one cache line per cycle, while the predicted path gets any left-over bandwidth ("pred-extra"), and a policy that gives the most fetch bandwidth to the path with the fewest number of instructions in the RUU ("pred-ruu", similar to a technique described in [30]. Fetch schemes favoring the predicted path require that the predicted path can be known and looked up at any time. This can be implemented using a set of per-path bitmasks similar to path IDs.

In [1], we found that giving more priority to the predicted path helps performance slightly— a few percent in most cases. Pred-extra outperforms the other two policies for all but one application. Among the SPECint95 benchmarks, *go* sees the least improvement from priority-based fetch schemes. This is because *go* has the least accurate branch prediction, so it is less likely to properly allocate fetch resources to the appropriate path.

4 Branch Prediction Issues

Because multipath execution attempts to reduce the penalties associated with mispredictions, accurate modeling of both prediction accuracy and timing effects are important. This section describes extensions in HydraScalar to the return-address stack, the modeling of branch-history update timing, and additional options for modeling different branch predictors.

4.1 Return-Address Stack

Return-address stack accuracy can be an especially strong lever on multipath performance. A single, unified stack does not function properly in a multi-path processor. With concurrent paths simultaneously modifying the stack, entries are popped and pushed by both correct and incorrect paths, making corruption almost certain. For example, after a fork, both paths might encounter calls to `printf()`. Both push a return address, even though only one return address belongs on the stack (only one call to `printf()` eventually commits). Neither mechanisms for repairing a return-address stack after mis-speculation [10, 22] nor per-path copies of the top-of-stack pointer can prevent this sort of corruption.

Per-path copies of the entire return-address stack are the best solution. Multipath execution already requires path contexts; the return-address stack is merely an additional element in the path context. Copying the stack should be no more expensive than saving and restoring the register map. A further consideration is that copying the stack need not take place in a single cycle; if the new stack only receives a correct top-of-stack pointer and value, the new path can begin popping or pushing, and deeper stack values can then be copied over in a more leisurely fashion.

HydraScalar also supports a unified stack with per-path top-of-stack pointers and a unified stack that only the predicted path can modify, but in [22] we found that these approaches are substantially inferior to per-path copies of the stack.

For uniprocessors, management of the return address stack is also important. Although we need only deal with one stack, a simple FIFO structure fails in the presence of speculative execution. The stack needs to be pushed and popped in the fetch stage—*i.e.*, speculatively—because otherwise return instructions will receive predictions based on a stale version of the stack. But only in the writeback stage do we know whether preceding branches have been correctly predicted. If not, speculative pushes and pops will have been invalid and have corrupted the stack.

Saving some shadow state with each in-flight branch solves this problem. The ideal solution would simply save and restore the entire return-address stack. Better yet, we have found that it is sufficient to save and restore only the top-of-stack contents and the pointer to the top-of-stack. Indeed, saving and restoring only the top-of-stack pointer [18] works reasonably well too. HydraScalar models all these solutions. These return-address-stack issues and solutions are described in more detail in [22].

4.2 Speculative History Update and Fixup

In addition to update timing for the return address stack, update timing also matters for the branch history in two-level predictors. This was shown by Hao, Chang, and Patt in [6]. A subsequent paper by Jourdan, Stark, Hsing, and Patt [10] also showed that update timing does *not* matter much for the table of two-bit counters, because the two-bit counters provide adequate hysteresis.

The problem for branch history is that if the history is updated when branches resolve at the end of the execution stage, a long time may elapse between the branch's prediction and its resolution. The branch must traverse the pipeline, and may in the meantime spend an arbitrary amount of time waiting for issue. In the meantime, subsequent branches see stale predictor state. For sequences of correlated branches, the global history may therefore show an inaccurate prior history that does not expose the correlation, and the correlated prediction is disrupted as a consequence. For individual branches with repeating patterns, the local history may show an inaccurate prior history that reflects the wrong point in the pattern, and the local-history prediction is disrupted.

Jourdan *et al.* describe some solutions for global history [10], and we extend this work and describe some solutions for local history in [24]. For both global and local history, the key is to update the branch history speculatively, as soon as the prediction is made. Of course, if execution is following a mis-speculated path, this will result in corrupted history. The solution for global history is to save and restore the global history value as shadow state for each in-flight branch. For local history, each modified location needs to be protected. Rather than saving and restoring many values in shadow state, an *outstanding branch queue* or OBQ can be used (also proposed in [20]). The table of per-branch histories (BHT) now stores only committed histories. Changes that are associated with in-flight branches go into the OBQ. When a branch commits, its OBQ entry is committed to the BHT. If it mispredicts, subsequent OBQ entries are simply discarded. New branch predictions must check both the BHT and the OBQ for the most up-to-date local history.

These approaches work because all instructions after the mispredicted branch are squashed. When they are re-executed, they see the corrected history. Of course, if there was no misprediction, the speculative history is in fact correct. HydraScalar supports both mechanisms.

4.3 Other Branch Modeling

HydraScalar includes other extensions to branch handling. To support modeling a finite amount of shadow state, the processor can be restricted to a maximum number of in-flight branches. To support various branch prediction schemes, a tagged, set-associative branch history table (BHT) can be modeled. Two new types of branch prediction are supported as well. In addition to the two-level/bimodal hybrid branch predictor [17], GAg, GAs, PAg, PAs [34], gshare [17], bimodal [27], and static predictors supported by SimpleScalar, HydraScalar adds a hybrid predictor with two different two-level prediction components, as described by Chang, Hao, and Patt in [3], and an *alloyed* predictor that combines global and local history in the same two-level structure, described in [26].

The new hybrid predictor allows the use of both global and local history. Chang *et al*. showed that it outperformed the global-history/bimodal organization proposed by McFarling, and we found the same to be true [25]. This is true because most programs have some branches that do well with global history *and* some branches that do well with local history [26].

Alloyed prediction also exposes global and local history, but it does so in an organization that resembles a conventional two-level predictor. This is done by concatenating the global- and local-history values and some branch-address bits before indexing the table of two-bit counters. Alloying is actually a generalization of *bi-mode* prediction [15] and can therefore be used to model that organization as well. In fact, alloying outperforms bi-mode prediction because bi-mode's "choice bits" are a restricted form of local history; alloying uses true local history and therefore incorporates more information.

5 Summary

This paper has described HydraScalar, which is made available under the same terms as the SimpleScalar license. HydraScalar is built on the SimpleScalar version 2.0 toolkit. It uses the same pipeline model as the toolkit's *sim-outorder* simulator, but the handling of speculative execution has been re-implemented to support the greater detail that the modeling of multipath execution requires. A variety of other extensions to sim-outorder are also included, such as more detailed modeling of branch-predictor update, new branch predictors, and additional pipeline detail. HydraScalar can be obtained by contacting the authors.

HydraScalar grew out of our research on multipath execution and branch prediction, and served as the simulation platform for a number of research projects. Some of these results have been briefly described here as explanation for the new features incorporated in HydraScalar. This work was supported in part by NSF grant CCR-94-23123, NSF Career Award CCR-95-02516, and an NDSEG Graduate Fellowship (Skadron). We wish to thank Doug Clark and Margaret Martonosi for their advice in development of the simulator.

References

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: Opportunities and limits. In *Proc. 12th ICS*, pages 101–08, July 1998.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[3] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proc. Micro-28*, pages 252–57, Dec. 1995.

[4] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, Feb. 16, 1995.

[5] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[6] E. Hao, P.-Y. Chang, and Y. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proc. Micro-27*, pages 228–32, Nov. 1994.

[7] T. H. Heil and J. E. Smith. Selective dual path execution. Tech. Report, Univ. of Wisconsin-Madison Dept. of Electrical & Computer Engineering, Nov. 1996.

[8] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *CompCon '95*, pages 123–28, Mar. 1995.

[9] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proc. Micro-29*, pages 142–52, Dec. 1996.

[10] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *Int'l J. Parallel Programming*, 25(5):363–83, Oct. 1997.

[11] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proc. Micro-32*, pages 38–47, Nov. 1999.

[12] A. Klauser, V. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath Architecture. In *Proc. ISCA-25*, pages 250–59, July 1998.

[13] R. Kol and R. Ginosaur. Kin: A high performance asynchronous processor architecture. In *Proc. 12th ICS*, pages 433–440, July 1998.

[14] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. ISCA-8*, pages 81–87, June 1981.

[15] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proc. Micro-30*, pages 4–13, Dec. 1997.

[16] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Computer Systems*, 15(3):322–354, Aug. 1997.

[17] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[18] S. McMahan, Cyrix Corp. Branch processing unit with a return stack including repair using pointers from different pipe stages. U.S. Patent No. 5,706,491, Jan., 1998.

[19] C. Price. *MIPS IV Instruction Set, Revision 3.1.* MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[20] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proc. ISCA-26*, May 1999.

[21] T. Santos, M. Pilla, R. Santos, E. C. Filho, S. Bampi, P. Navaux, and M. Nemirovsky. Resource tuning in a multipath superscalar architecture. In *Proceedings of the 11th Symposium on Computer Architecture and High-Performance Computing*, Sept. 1999.

[22] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proc. Micro-31*, pages 259–71, Dec. 1998.

[23] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Trans. Computers*, 48(11):1260–81, Nov. 1999.

[24] K. Skadron, D. W. Clark, and M. Martonosi. Speculative updates of local and global branch history: A quantitative analysis. *J. Instruction-Level Parallelism*, Jan. 2000. (http://www.jilp.org/vol2).

[25] K. Skadron, M. Martonosi, and D. W. Clark. Alloying global and local branch history: A robust solution to wrong-history mispredictions. Tech. Report TR-606-99, Princeton Dept. of Computer Science, Oct. 1999.

[26] K. Skadron, M. Martonosi, and D. W. Clark. A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In *Proc. PACT 2000*, Oct. 2000. To appear.

[27] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA-8*, pages 135–48, May 1981.

[28] G. S. Sohi and A. S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proc. ISCA-14*, pages 27–34, June 1987.

[29] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. WWW site: http://www.specbench.org/osg/cpu95, Dec. 1999.

[30] D. Tullsen, S. Eggers, J. Emer, et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. ISCA-23*, pages 191–202, May 1996.

[31] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Tech. Report CSE-TR-346-97, Univ. of Michigan Dept. of Electrical Engineering and Computer Science, 1997.

[32] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proc. Micro-28*, pages 313–25, Dec. 1995.

[33] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proc. ISCA-25*, pages 238–49, July 1998.

[34] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. ISCA-19*, pages 124–34, May 1992.