Traveling Speculations: An Integrated Prediction Strategy for Wide-Issue Microprocessors

Ravi Bhargava, Juan Rubio and Lizy K. John

Technical Report TR-020524-01
Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
{ravib,jrubio,ljohn}@ece.utexas.edu

May 2002

Abstract

Performing multiple, accurate, low-latency predictions is crucial to improving instruction throughput in future wide-issue microprocessors. However, demands of wide-issue processing coupled with implementation challenges posed by high clock frequencies present obstacles to these prediction goals. This paper proposes the Traveling Speculation framework to accommodate predictions in a wide-issue environment. Instead of using large centralized data tables to perform predictions, distributed predictors are introduced along with the Traveling Speculation hint, or *tint*. A tint contains execution history and is assigned to each instruction that is in-flight or in the trace cache. The tint can be related to branch behavior, cache behavior, value prediction, or any history-based aspect of microarchitecture. By associating a unique tint with each instruction in the trace cache, a per-instruction, interference-free execution history is maintained, providing superior prediction accuracy and instruction coverage in a low-latency manner. Our results show that the Traveling Speculation framework improves prediction accuracy and provides personalized execution history to a high percentage of dynamic instructions. Value prediction using Traveling Speculation tints improves instruction throughput by 9.7% over the base model, in comparison to 5.0% obtained by a table-based value predictor. Similarly, speculative memory forwarding using tints improves instruction throughput by 11.8% compared to 7.3% obtained using a memory renaming scheme.

1 Introduction

Improving instruction throughput continues to be a high priority for high-performance microprocessors. Increasing issue width or resources such as the instruction window, functional units, and cache does not always result in sufficient improvement, especially for irregular applications. Fundamental program characteristics such as data and control dependencies limit opportunities for high instruction throughput. However, studies show that accurate, low-latency predictions coupled with aggressive instruction fetching can improve instruction throughput well beyond current levels [10, 11, 19].

Several aspects in the state-of-the-art microprocessor's operation are guided by accumulated histories. For instance, branch prediction, value prediction, cache replacement, and advanced cluster

scheduling are all governed by information collected by history-capturing structures. The histories are typically stored in one or more tables where they can be accessed and updated. Branch history tables, branch target buffers, and value prediction tables are examples of structures that are based on capturing instruction execution history.

The latency to access large structures is becoming an obstacle to achieving high instruction throughput. This is the result of history tables growing in area and complexity, while cycle times become more wire-delay dominated, [1, 14]. Prediction tables have become large to avoid destructive interference (multiple instances corrupting one particular table entry). The latency problem is more serious if a table is not located in close proximity to the pipeline stage that accesses its data. In some situations, a multiple cycle latency is tolerable, especially if the prediction can be pipelined easily. However, the extra latency is often undesirable and can lead to minimal performance gain or no gain at all.

Increasing the number of read and write ports of the tables aggravates the latency problem. For instance, Figure 1 contains latency in clock cycles for a 32kB table with different read port configurations using Cacti 2.0 [32]. Reducing ports limits the coverage of aggressive speculation, as one-ported or two-ported structures do not offer the sufficient access bandwidth needed by a wide-issue environment. Consider that several basic blocks of instructions are introduced to the wide-issue processor in each cycle and potentially hundreds of instructions are in-flight at any given time. Therefore performing multiple predictions per cycle is very attractive.

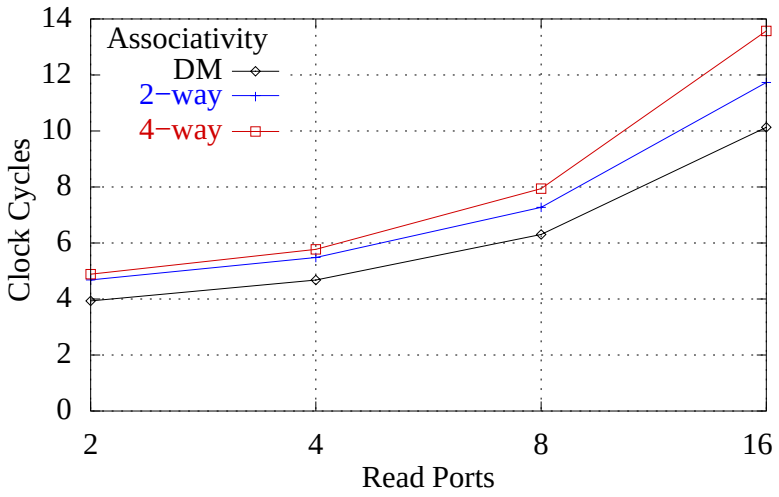

Figure 1: Clock Cycles to Access a 32KB Buffer with Varying Read Ports
Absolute latencies are obtained using Cacti 2.0 for a 100nm technology, 64-bit data blocks, one read/write port, two write ports, and a varying number of read ports. A clock frequency of 3.5 GHz is chosen based on the SIA road map project for a 100nm process [36]

To solve the aforementioned problems of centralized prediction buffers, we present the *Traveling Speculation* framework, where predictors are several distributed components instead of a centralized

table-driven mechanism. Prediction information travels with the instructions through the processor pipeline. The prediction logic, update logic, history for in-flight instructions, and history storage for retired instructions are physically separated. The binding element is the traveling speculation hint, or the *tint*, which contains execution history, prediction hints, and profiling information. A high-level view of the proposed scheme is shown in Figure 2. Each in-flight instruction has a corresponding tint, which is present in locations such as the decoder, dispatcher, reservation stations, reorder buffer, etc. Trace caching, which is present in modern commercial processors [12], presents a convenient method for the storage of the tints. When a new trace cache line is built, a corresponding trace of tints is stored. At any time, there exists a one-to-one mapping between instructions in the trace cache and the stored tints. When the trace cache is accessed, the instructions and associated tints are acquired simultaneously.

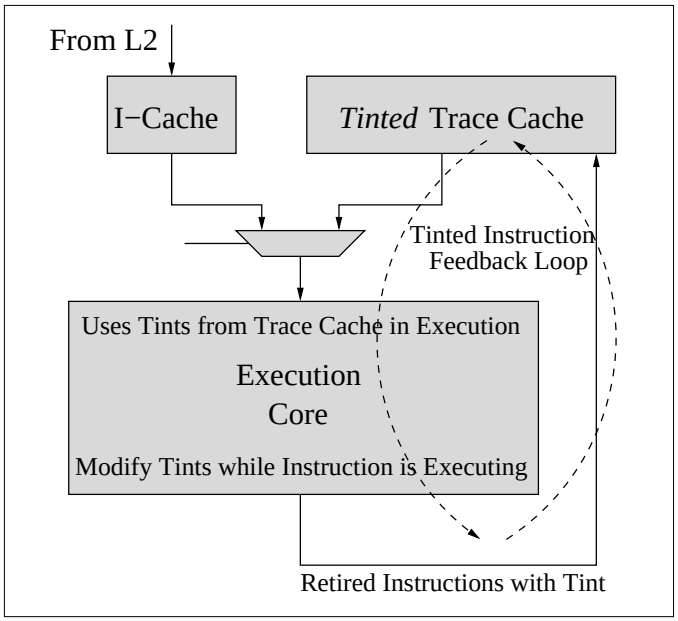

Figure 2: Overview of the Traveling Speculation Framework

Depending on the target applications and the specific bottlenecks of the processor in question, the Traveling Speculation framework can be used to improve prediction techniques or microarchitecture policies based on accumulating history, such as value prediction, control prediction, power optimization opportunities, cluster scheduling policies, and replacement policies. Instead of viewing the prediction process as an awkward appendage to the microarchitecture, we consider history-based tuning of the processor as an essential component of high-performance execution. Therefore, we integrate execution histories into the instruction flow path. Our approach exploits the transistors and localized on-chip bandwidth readily available on future microprocessors, while avoiding high-latency communication. The use of per instruction, per path (context) tints improves prediction accuracy

and instruction coverage, while eliminating many of the accesses to centralized prediction tables.

This work analyzes the Traveling Speculation framework in the context of a general-purpose processor executing SPEC2000 integer applications. Using value prediction and speculative memory forwarding tints as Traveling Speculation examples, we show high instruction coverage, wide prediction bandwidth, and improved prediction accuracy. We also analyze the benefits of storing the tints in a trace-based format. When compared to table-based structures, Traveling Speculations yield 4.7% more performance over our base model for versus a table-based value predictor. The improvement is 4.5% using a speculative memory forwarding tint versus memory-forwarding tables.

The next section describes the basics of the Traveling Speculation framework and discusses two applications of the Traveling Speculation framework. Section 3 describes the benchmarks and simulation environment. In Section 4, we analyze when and how the Traveling Speculation framework provides beneficial performance for a Value Prediction tint and Speculative Memory Forwarding tint. We compare these results to table-based value predictor and memory forwarding strategies, respectively. Related work is presented in Section 5. Section 6 summarizes the paper and presents our conclusions.

2 Description of the Traveling Speculation Framework

Traditionally, predictors are treated as stand-alone mechanisms separate from the execution core. The prediction information is typically held in a cache-like table, indexed by some distinct characteristic of the instruction, such as its program counter bits. This approach is well understood and common-place in microarchitecture. However, this strategy can impose limitations on the accuracy and coverage of predictions due to destructive interference and limited table entries.

The Traveling Speculation framework takes a different approach to the process of prediction. The goal is to provide high-bandwidth, low-latency predictions while increasing prediction accuracy and instruction coverage. The process of prediction is distributed throughout the microarchitecture using a per instruction traveling hint, or *tint*, as the binding element. The tint data can contain microarchitecture state that the instruction encounters while executing, specialized prediction hints, and dynamic profiling information. Using branch prediction as an example, a possible tint may have a taken/not taken history (microarchitecture state), a confidence state (prediction hint), and a misprediction counter (profiling information). In a traditional microarchitecture, the contents of a tint would be stored in a table, if they are stored at all.

In the Traveling Speculation framework, the tint data is constantly available with the in-flight instruction, eliminating a table access. Between invocations of the instruction, the tint data is stored

using a one-to-one mapping with the instructions stored in the trace cache. The traveling speculation approach fits well into the wide-issue environment supported by a trace cache. A trace cache provides high instruction bandwidth fetches by storing dynamic traces of execution that can contain multiple basic blocks [2, 7, 29, 33]. The Traveling Speculation framework takes advantage of the trace cache instruction feedback cycle, where retired instructions are written to and fetched from the trace cache. A detailed view of a Traveling Speculation framework is shown in Figure 3. The arrows represent the path of instructions and shaded areas represent possible points at which the tints are present and active.

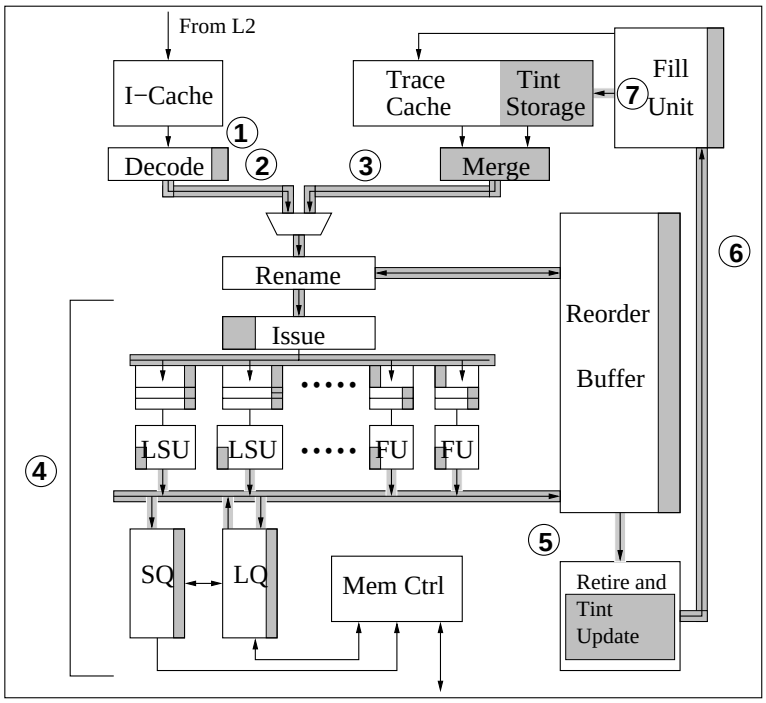

Figure 3: Detailed Look at a Traveling Speculation Framework
Shaded areas are possible points for tints to be active

A description of the various destinations of a Traveling Speculation follows below and corresponds to Figure 3.

1. An instruction must first be fetched from the instruction cache into the processor and then decoded. At this point, it does not have any tint. item As the instruction is decoded, the tint is initialized based on its instruction type and injected into the instruction's micro-data (necessary per instruction microarchitecture information for executing and retiring the instruction).

2. If a trace cache hit occurs, the instructions and tints associated with that trace cache entry are fetched in parallel. The tints are accessed using the same indexing method as the trace cache.

Because there is a one-to-one association between the instruction storage and tint storage, the fetching of personalized tints comes relatively easily. This is the only time tints are read from the tinted portion of the trace cache.

3. The tint remains associated with the instruction wherever it is needed - issue logic, reservation stations, functional units, reorder buffer, memory controller, etc. At these points, the tint can be used to guide predictions.

4. After the microarchitecture is finishes executing the instruction, the tint is altered based on the execution circumstances and results.

5. As the instruction is retired, its tint is also sent to the trace cache fill unit, which is in charge of creating and storing the dynamic instruction traces.

6. When the fill unit has created a new trace, it writes the tint to the trace-based tint storage.

Using trace-based storage provides the tints with a degree of natural path correlation in addition to the inherent per instruction correspondence. History in prediction tables is usually accessed using the program counter (PC) of the corresponding instruction. In addition to instruction uniqueness, an instruction's setting within the dynamic flow of the program is important. Previous research shows that capturing information along the path that leads to an instruction, combined with the PC, can provide a more distinguishable behavior pattern than using just the PC [25, 26, 27, 40, 41]. The tints are stored in trace format, providing the associated history with some context.

Traveling Speculations also provide the opportunity to dynamically acquire per instruction information, a task sometimes relegated to static profiling. It has been demonstrated that many predictors are more effective if the run-time behaviors of the target applications are understood before execution using pre-execution profiling [16, 25, 31, 37]. However, this type of profiling requires multiple program runs and is often imprecise due to the large variance in data input sets and program phases. Transferring the profile information to the individual instructions is tricky and can involve instruction set architecture alterations. Also, the supporting compiler is complicated and requires detailed hardware knowledge. The Traveling Speculation framework has the ability to capture personalized history dynamically and feed it back to the microarchitecture at run-time.

2.1 Value Prediction Tint Example

This work presents a value prediction tint (VP-Tint) as an example application of the Traveling Speculation framework. The predicted value, stride, and confidence counter are all stored in the

VP-Tint. Since the prediction data is self-contained, value prediction using the VP-Tint does not require any prediction tables. Only eligible instructions (integer loads and arithmetic) are given an active tint, and the prediction mechanism works exactly the same as table-based stride prediction [9]. Our example microarchitecture with a VP-Tint tint is presented in Figure 4.

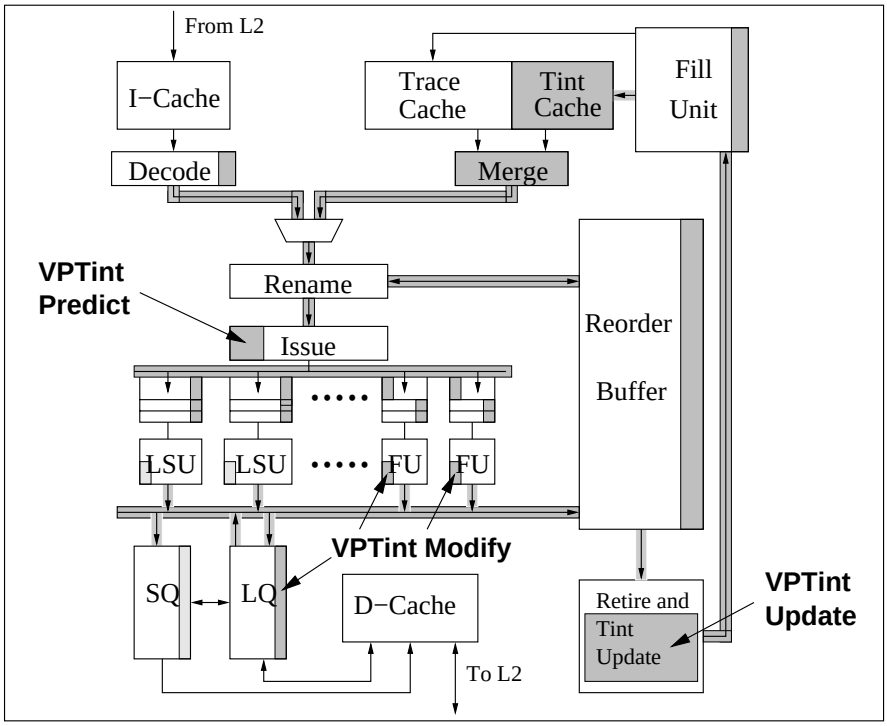

Figure 4: Traveling Speculation Framework with the Value Prediction Tint

The VP-Tint uses the Traveling Speculation framework illustrated in Figure 3. An eligible instruction receives a VP-Tint once it is decoded on the instruction cache fetch path (point 2 in Figure 3). The VP-Tint value, stride, and confidence fields are all set to an initial value. If the instruction is fetched as part of a trace from the trace cache (point 3), the existing VP-Tints are incorporated into the instruction micro-data after the fetch. Value predictions occur after the instructions are decoded and before they reach the instruction window (point 4 at issue). A prediction is executed only if the VP-Tint confidence counter demonstrates the proper confidence. If this is the case, the VP-Tint value is speculatively assigned to the instruction's destination register so that dependent instructions can benefit from the prediction (as in standard value prediction). Once the instruction is executed and reaches a non-speculative state, the prediction is verified and the VP-Tint fields are updated before being processed by the Fill Unit.

2.2 Speculative Memory Forwarding Tint Example

The speculative memory forwarding tint (SMF-Tint) is another application of the Traveling Speculation framework. It has been shown in the past that a large fraction of load instructions obtain data recently written by a store instruction from the same program. The purpose of this tint is to allow load instructions to speculatively receive data from instructions that store to the same memory location before the address of the load has been calculated. A load that speculates correctly frees dependent instructions early instead of waiting to access the memory subsystem. The SMF-Tint accomplishes something similar to previous memory forwarding hardware [24, 30, 39].

The SMF-Tint is different from the VP-Tint because it uses accompanying tables as shown in Figure 5. Load instructions speculatively obtain values from the value file, which holds the data or data tag from a previous store instruction. The store cache identifies store-load dependencies non-speculatively and organizes the speculative indexing into the value file. The functionality of the store-load cache from memory renaming is replaced by the SMF-Tint, so only one table needs to be accessed at prediction time. The tables for the memory renaming hardware function as described in [31, 39]. While the use of a value file table does not allow this particular Traveling Speculation to take full advantage of the prediction bandwidth and latency properties of the framework, the framework still offers performance boosts through an increase in prediction accuracy and instruction coverage.

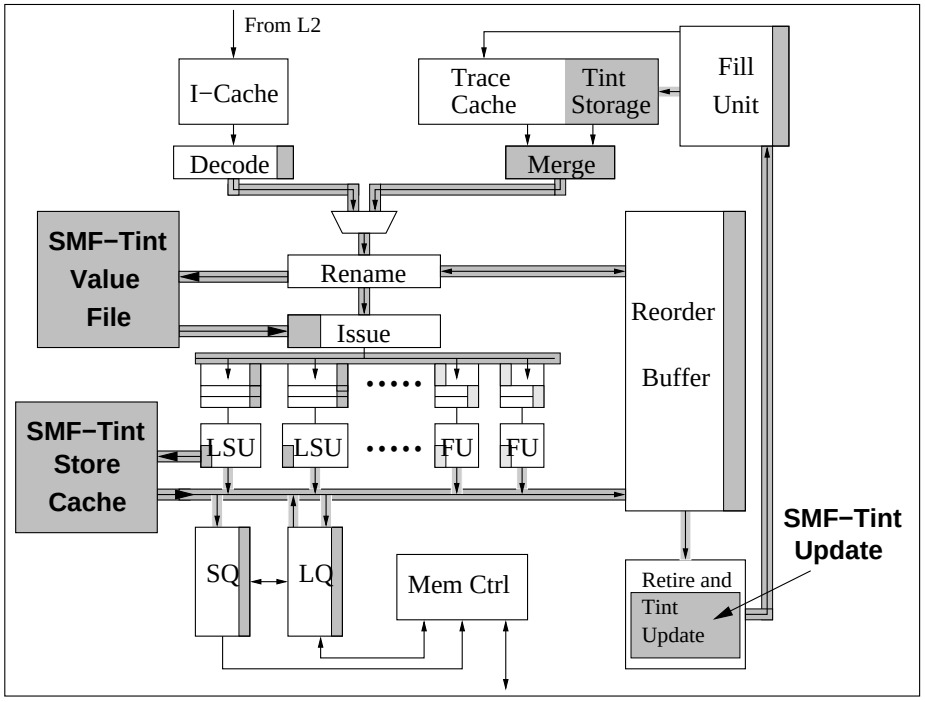

Figure 5: Traveling Speculations: Speculative Memory Forwarding Tint

The SMF-Tint has three fields for store instructions and two fields for load instructions. The interaction of the SMF-Tint with the store cache and value file is shown in Figure 6. The store SMF-Tint contains a filter for the value file, a `HasIndex` field, and an index field. This filter is a two-bit saturating counter that prevents a store instruction from writing to value file entries if it is determined that it is not forwarding data to load instructions. The `HasIndex` field is a one-bit field that works like a valid bit. If the field is set to one, then this store has been assigned a proper index and can access the value file. The load SMF-Tint contains a one-bit confidence counter and an index. The confidence counter is based upon the confidence of the prediction accuracy.

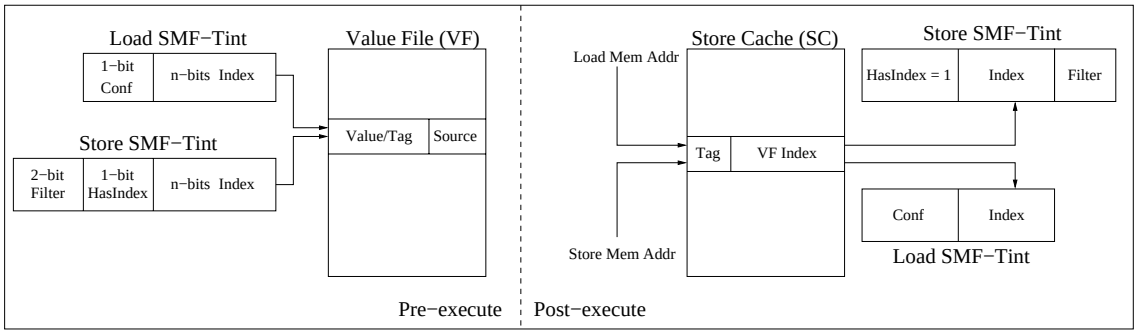

Figure 6: Interaction of Load and Store SMF-Tint with the Tables

2.3 Summary of Traveling Speculations

Traveling Speculations have several unique attributes that provide significant advantages over traditional table-based prediction. The per instruction nature of the tints permits execution histories to be truly unique. For example, the VP-Tint behaves like a fully-associative prediction table where the lack of destructive interference makes the predictions very accurate. A table is limited to a fixed number of ports, but the VP-Tint is equivalent to an infinite port mechanism where each instruction can obtain a prediction.

Tints can be used to dynamically profile the run-time instruction behavior by providing per instruction counters. Strategies have been proposed that use compiler and/or static profiling help for prediction hardware. All of these features are a product of the readily available local bandwidth instead of long-distance communication and data access latency, making this an attractive implementation for present and emerging processing technologies. For example, the filtering done by the SMF-Tint uses per instruction execution history to improve the access efficiency of the value file.

Using the Traveling Speculation framework, the latency to access an instruction's personalized history is less susceptible to multi-cycle on-chip latencies because the tint is part of the instruction micro-data. Furthermore, self-contained tints (those that do not require tables) provide the equiva-

lent of an infinite-ported structure. Most encouragingly, once the Traveling Speculation framework is implemented for one particular tint, it is inherently suitable for a variety of tints. On an example by example basis, there seems to be wasted tint storage space since all instructions are not tintable. However, in its full glory all classes of instructions are tinted with at least one tint. The framework is not specific to value prediction or memory forwarding. Instead, these are only examples of how Traveling Speculation can readily improve upon prediction and history-based techniques.

3 Methodology

In this section, we describe the benchmarks and simulation environment used to study the Traveling Speculation framework. We examine the integer benchmarks from the SPEC CPU2000 suite [38]. Because the `test` input for `perlbmk` provides a very short run, a modified `ref` input is used. With these inputs, the run length of this benchmark matches the run length for other benchmarks using the test inputs. The SPEC programs are dynamically linked and compiled using the Sun Compiler with the `-fast -xO5` optimizations. The benchmarks and their respective inputs are presented in Table 1.

Table 1: SPECINT2000 Benchmarks

Benchmark	Inputs	Instr Simulated
bzip2	input.random 2	300M
crafty	crafty.in	300M
eon	chair.control.kajiya chair.camera chair.surfaces chair.kajiya.ppm ppm pixels_out.kajiya	300M
gap	-l ./ -q -m 64M test.in	300M
gcc	cccp.i -o cccp.s	300M
gzip	input.compressed 2	300M
mcf	inp.in	251M
parser	2.1.dict -batch test.in	300M
perlbmk	-I. -I./lib splitmail.pl 1 5 19 18 150	300M
twolf	test	243M
vortex	bendian.raw	300M
vpr	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	300M

To perform our simulation, we use a functional, program-driven, cycle-accurate simulator. Given a SPARC executable as input, the front-end functional execution is performed by the Sun tool Shade [3]. The simulator models many of the modern aggressive instruction-level parallelism techniques, as well as all resource contentions and speculative execution. We do not "idealize" any portion of the microprocessor since the Traveling Speculation framework is best analyzed using a realistic balance of resources. The basic pipeline is six stages: fetch, decode/merge, rename, issue, execute, and write-back. Memory operations require additional pipeline stages, including TLB access and cache

access. The parameters for the simulated base microarchitecture can be found in Table 2.

Table 2: Simulated architecture parameters

Data memory	
· L1 Data Cache:	4-way, 32KB, 2-cycle access
· L2 Unified cache:	4-way, 1MB, +8 cycles
· Non-blocking	12 MSHRs and 2 ports
· D-TLB	512-entry, 4-way, 1-cycle hit, 30-cycle miss
· Store buffer:	64-entry w/load forwarding loads access in 1-cycle
· Load queue:	64-entry, no speculative disambiguation
· Main Memory	Infinite, +65 cycles

Fetch Engine	
· Trace cache:	4-way, 2K entry, 1-cycle hit partial matching, no path assoc.
· L1 Instr cache:	4-way, 4KB, 1-cycle hit one basic block per access
· Branch Predictor:	16k-entry gshare/bimodal hybrid predictor 3-cycle misprediction penalty + refetch
· Branch target buffer	512 entries, 4-way

Execution Core

· Functional unit	#	Exec. lat.	Issue lat.
Load/store	8	1 cycle	1 cycle
Simple Integer	8	1	1
Int. Mul/Div	2	3/20	1/19
Branch	4	1	1
Simple FP	4	3	1
FP Mul/Div/Sqrt	1	3/12/24	1/12/24

· Data Forwarding Latency: 1 cycle
· Register File Latency: 2 cycle

· 256-entry INT ROB; 128-entry FP ROB
· 12 reservation station entries/INT func. unit
· 16 reservation station entries/FP func. unit
· Machine width: 16 instructions

The distribution of the functional units favors general-purpose integer benchmarks. There is a one-cycle latency for data forwarding to another functional unit and two-cycle latency for accessing the register file. A 2048-entry trace cache is accompanied by a small 4kB instruction cache. The trace cache is modeled after the trace cache in [8]. In our model, partial hits are allowed, but path associativity is not. The fill buffer writes a new trace cache line each time a new one is created, regardless of whether a similar trace is already present. The multiple branch predictor performs the equivalent of three hybrid predictions [20] per cycle. Trace cache writes do not take place simultaneously with a read to the same set. Store buffer forwarding takes one cycle.

For various forms of data prediction (value prediction, memory renaming, VP-Tint, SMF-Tint), the aggressive *selective reexecution* mechanism is used. Only instructions in the dependency chain of the mispredicted instruction are restarted. These instructions are placed into a reissue queue after a fixed latency (three cycles). The reissue queue can accept 16 instructions per cycle and has priority over the correct path issue queue. These same predictions are not done on reissued instructions.

4 Evaluation of the Traveling Speculation Framework

In this section we discuss the effectiveness of the Traveling Speculations. We look at the instructions retired per cycle, and analyze the speedup provided by the value prediction tint and speculative memory forwarding tint. The performance differences are also compared to known table-based methods.

4.1 General Analysis of the Traveling Speculation Framework

Table 3 presents some trace cache characteristics for the base model which are important to the effectiveness of the Traveling Speculation framework. These results are presented first to assist the interpretation of the performance results.

Table 3: Base Model Trace Cache Characteristics

	bzip2	cc1	crafty	eon	gap	gzip	mcf
TC Instr Pct	98.64	86.06	92.36	97.00	97.53	99.81	99.77
TC Conflict Pct	0.00	9.23	1.65	0.04	0.42	0.00	0.00
TC Rd Wr Conf Pct	21.77	5.34	0.54	3.77	0.83	11.96	10.07

	parser	perlbmk	twolf	vortex	vpr	Mean
TC Instr Pct	90.27	91.86	89.86	93.12	97.92	94.31
TC Conflict Pct	0.32	0.00	3.45	1.64	0.00	0.21
TC Rd Wr Conf Pct	4.00	2.09	0.75	2.16	0.39	1.35

TC Instr Pct: Percentage of fetched instructions - correct or wrong path - that come from the trace cache.
TC Conflict Pct: Percentage of trace cache writes that cause a set replacement.
TC Rd Wr Conf Pct: Percentage of trace cache reads that cause a trace cache write to be delayed.
Mean: The mean is a harmonic mean.

An important statistic for understanding the performance numbers in this paper is the percentage of fetched instructions that come from the trace cache (`TC Instr Pct`). If instructions are not fetched from the trace cache, then they have no associated tints. Therefore, good trace cache hit rates are preferable. Overall, 94% of instructions that are decoded come from the trace cache. This ranges from 86% in the `cc1` benchmark to almost 100% in `gzip` and `mcf`. Effectively, this is the instruction coverage provided by Traveling Speculations since each instruction fetched from the trace cache has an associated tint with current history information.

Most of the misses and resulting I-Cache fetches are due to cold misses and capacity misses. `TC Conflict Pct` is the percentage of trace cache writes that actually cause a set replacement. This number is below 1% for eight of the 12 benchmarks. Only `cc1` shows significant set replacement. Our studies indicate that the trace cache misses for `cc1` are due to new regions of code rather than the thrashing of interfering traces.

Though infrequent, the scenario exists where a trace is discarded from the trace cache and recreated later. In this case, the history accumulated in the tint is lost and must be acquired once again. A table-based mechanism has the opportunity to retain a history in this scenario. However, if the instruction footprint is large and exhibits random access patterns that would thrash a trace cache, it is also likely to cause destructive interference in a prediction table.

The final percentage in Table 3 is the percentage of trace cache reads that delay a trace cache write (`TC Rd Wr Conf Pct`). In our simulations, the fill unit cannot write to the trace cache in the

same cycle as a read access to the same set. This is more relevant for the tints than the instruction traces. When a trace is formed that is identical to a current trace, no new information is written to the instruction storage of the trace cache. However, for each trace captured, new history needs to be written to the tint storage. If the extra tint writes are consistently conflicting with tint reads, then the tinted information update is delayed or even discarded. However, the table shows that the read-write conflict percentage is fairly low. Over the SPECINT2000 benchmarks, only 1.3% of reads cause a write to be delayed.

4.2 Analyzing the Value Prediction Tint

Figure 7 summarizes the performance of the VP-Tint. Speedups versus the base model are presented for the VP-Tint and two versions of table-based stride value predictors. The mean is presented as a geometric mean throughout the results unless noted. Stride prediction is done with a basic stride predictor [9] with 4096 entries. Although there are more complex methods of value prediction [18, 35, 40], we study a large stride predictor because it is more easily realizable and captures both the striding and last value behaviors. The 16 port stride predictor presented in the figure (`Stride VP Port 16`) can accommodate up to 16 predictions and updates each cycle. The second stride predictor presented (`Stride VP Port 2`) has a more realistic implementation, using two read ports, but still permitting 16 updates. In this case, the first two eligible instructions (memory and integer operations that write a destination register) are predicted on a "first-come, first-serve" basis, and instructions get one opportunity to access the table. The access latency for both predictors is just one clock cycle, although this latency could be multiple clock cycles in future technologies.

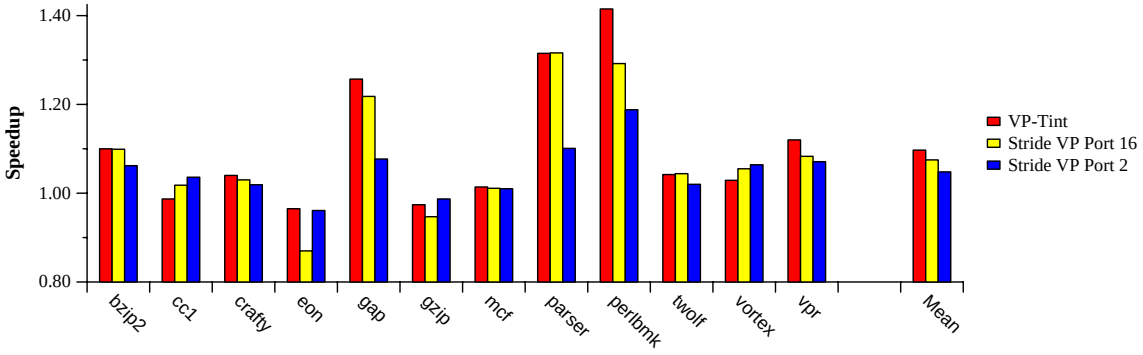

Figure 7: Value Prediction: Speedups Versus the Base
`VP-Tint`: as described in the paper.
`Stride VP Port 16`: 4096-entry table, 16 read ports, 16 write ports, 1-cycle latency.
`Stride VP Port 2`: 4096-entry table, two read ports, 16 write ports, 1-cycle latency.

Figure 7 illustrates that the value prediction Traveling Speculation improves performance by 9.7% over the base model for SPECint2000. This method of prediction outperforms a large, low-

latency, 16 port stride value predictor by almost 2%. However, the table-based stride predictor has been generously provided with 16 read and write ports, as well as a one-cycle access latency. This combination of parameters would not be possible in future high-performance, high-frequency processors. When the stride predictor is restricted to two read ports, the performance contribution over the base provided by the VP-Tint is almost double that of the stride predictor, 9.7% to 5.0%.

Individual benchmarks provide some interesting results. For example, several benchmarks show better stride predictor performance when port-limited (`cc1, eon, gzip, vortex`) than with 16 ports. The precise reason is difficult to isolate since performance benefits of value prediction are based on a combination of factors, such as misprediction rate, prediction confidence, update frequency, varying rewards for correct predictions, and varying penalties for mispredictions [18, 34].

Table 4: Analysis of Value Prediction Effectiveness

	bzip2	cc1	crafty	eon	gap	gzip	mcf
VP-Tint MP Rate	7.57	7.84	4.03	8.89	5.37	5.55	17.51
SVP-2 MP Rate	13.00	13.57	5.21	16.26	8.41	8.26	29.27
VP-Tint Coverage	98.43	84.20	94.64	95.98	95.56	99.93	99.71
SVP-2 Coverage	99.18	77.42	69.36	82.39	77.28	98.78	99.34

	parser	perlbmk	twolf	vortex	vpr	Mean
VP-Tint MP Rate	7.91	7.02	5.03	3.26	6.44	6.08
SVP-2 MP Rate	28.45	9.22	7.89	9.68	8.86	10.30
VP-Tint Coverage	96.56	94.22	87.82	90.93	97.44	94.38
SVP-2 Coverage	95.49	86.75	86.23	65.81	93.34	84.39

`VP-Tint`: as described in this paper
`SVP-2`: two read port stride value predictor
MP Rate: misprediction rate = retired loads predicted incorrectly / retired loads predicted
Coverage: percentage of instructions that have an available history to perform prediction
Mean: harmonic mean of the benchmark suite

Table 4 contains some of the prediction-related characteristics for the VP-Tint and two read port stride predictor. Over all of the programs, the VP-Tint hardware provides a significantly lower misprediction rate (`MP Rate`) than the table-based methods, 6.08% versus 10.30% for the two port stride value predictor. Often in value prediction schemes, the total number of correct predictions must be sacrificed for a higher accuracy. However, even with the superior accuracy, the VP-Tint still provides 2.75 times the number of correct predictions as the two read port predictor (0.94 times as many correct predictions for than a 16 port stride predictor).

The lower section of the table presents the instruction coverage (`Coverage`) for the two read port stride value predictor and the VP-Tint. This represents the percentage of instructions that have an available history. For the stride value predictor, this results from a tag-match in the table. For Traveling Speculations, this means any instruction fetched from the tinted trace cache. The results in the table show that the Traveling Speculations do a better job of providing per instruction history

than the 4k-entry, direct-mapped, table-based predictor.

Two programs show better performance using table-based prediction than the VP-Tint (`cc1`, `vortex`). In these programs, correct predictions do not have a high reward. So even though the two read port stride value predictor has a worse prediction rate and fewer correct predictions, it outperforms the VP-Tint because it has the fewest mispredictions.

4.3 Analyzing Speculative Memory Forwarding

The SMF-Tint is analyzed versus the base model and compared to a version of the memory renaming hardware in [31, 39]. The memory renaming hardware consists of a 4-way, 1024-entry store cache, a 1024-entry value file, and a 4-way, 1024-entry store-load cache. The store cache and value file have similar functionality to the VP-Tint store cache and value file. The SMF-Tint has the equivalent purpose of the store-load cache, resulting in one fewer table. The process of accessing the two levels of tables in the memory renaming scheme (store-load cache and value file) has a one-cycle access latency. The speedups versus the base model for the different speculative memory forwarding strategies are presented in Figure 8.

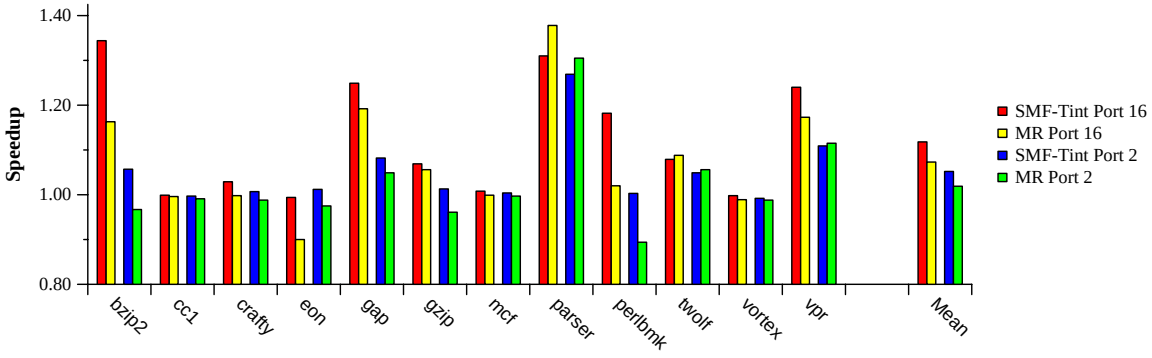

Figure 8: Speculative Memory Forwarding: Speedups Versus the Base
`SMF-Tint Port 16`: SMF-Tint with 16 ports to value file
`MR Port 16`: memory renaming with 16 read ports to the store-load cache and value file
`SMF-Tint Port 2`: SMF-Tint with two read ports to value file
`MR Port 2`: memory renaming with two read ports to the store-load cache and value file

The first two bars for each benchmark in the figure represent the SMF-Tint with a 16 read port value file versus memory renaming hardware with a 16 read port store-load cache and value file. Using the SMF-Tint as a replacement for the store-load cache results in 11.8% improvement over the base model versus 7.3% using memory renaming, and in all cases except for `twolf`, the SMF-Tint provides an improvement versus the table-based method. As seen in the two right-hand bars, when placing a two read port restriction on both sets of hardware (value file for SMF-Tint, store-load cache and value file for memory renaming) while keeping 16 write ports, the SMF-Tint continues to provide superior performance, 5.2% versus the base model for the programs examined versus 1.9%

using memory renaming.

Table 5 presents the misprediction rate and coverage for the memory forwarding strategies. The advantage of the SMF-Tint is a much lower misprediction rate (`MP Rate`). The misprediction rate using the SMF-Tint with 16 ports to the value file is only 11.30% versus 24.26% for the memory renaming hardware. Although the memory renaming hardware is able to produce 14% more correct predictions than the SMF-Tint, the increase in mispredictions overshadows this advantage. The instruction coverage when using the Traveling Speculations is part of the reason for the improved misprediction rate. The lower portion of the table shows the load coverage for the two mechanisms. A greater percentage of loads has an available instruction history in the Traveling Speculation framework than in the strictly table-based case.

Table 5: Analysis of Speculative Memory Forwarding Effectiveness

	bzip2	cc1	crafty	eon	gap	gzip	mcf
SMFT-16 MP Rate	3.65	16.43	11.70	33.29	17.82	15.97	22.18
MR-16 MP Rate	39.40	19.41	17.51	63.96	24.37	23.93	44.45
SMFT-16 Coverage	99.17	83.47	93.42	96.20	97.79	99.84	99.64
MR-16 Coverage	99.99	88.43	71.82	63.00	98.66	99.99	99.99

	parser	perlbmk	twolf	vortex	vpr	Mean
SMFT-16 MP Rate	12.82	5.70	19.65	11.75	17.27	11.30
MR-16 MP Rate	15.49	37.45	26.05	12.68	32.19	24.26
SMFT-16 Coverage	93.41	91.910	89.670	91.88	97.79	94.27
MR-16 Coverage	96.14	94.750	91.130	74.54	98.94	87.78

`SMFT-16`: SMF-Tint with 16 read ports to value file
`MR-16`: memory renaming with 16 read ports to the store-load cache and value file
MP Rate: misprediction rate = retired loads predicted incorrectly / retired loads predicted
Coverage: percentage of loads that have an available history to perform prediction
Mean: harmonic mean for benchmark suite

4.4 Correlation Analysis

The Traveling Speculation framework provides high instruction coverage, low-latency accesses, and high-bandwidth prediction capability. The instruction coverage and prediction bandwidth have been discussed in the previous sections while the low-latency access capability is not directly quantified in this work. However, one more characteristic of the Traveling Speculation framework that works to the advantage of the studied prediction mechanisms is path correlation.

Path correlation is present because tints are stored in a trace format. The same static instruction may be present multiple times in the trace cache but in the context of different traces. This has the advantage of providing a more distinctive execution history for speculation, but has the potential disadvantage of diluting the instruction history for a particular static instruction. To study the correlation effect, we compare the Traveling Speculation framework described in Section 2 with a

Traveling Speculation framework with a different tint storage. The new storage scheme stores one tint per PC in an infinite sized table. When a trace is fetched from the trace cache, this table is accessed once per instruction to obtain a strictly per PC history without any correlation effects. Obviously, this exact mechanism for tint storage is infeasible, but presents an upper-bound for precise per instruction storage.

Figure 9 presents IPC speedups for the two methods of tint storage for both the VP-Tint and SMF-Tint. In both cases, the trace-based storage provides better performance than per PC storage. The difference is not as great in the case of the VP-Tint, but remember that the per PC storage is of infinite size and never loses history while the per trace storage can lose history when a trace is evicted from the trace cache. Still, using trace-based storage for the SMF-Tint proves to be a much better method of providing meaningful per instruction histories than the per PC method. This is another reason that the SMF-Tint performs well versus the table-based renaming scheme.

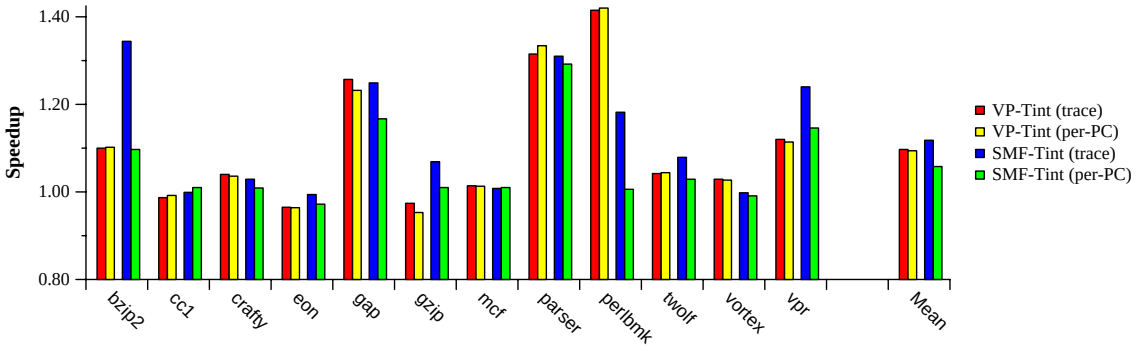

Figure 9: Speedups for Path Correlated Tint Storage and Per PC Tint Storage

5 Related Work

The concept of caching and refetching instructions with extra information has been discussed in various forms. Trace caches store information beyond the decoded instruction information. Traditional versions of the trace cache choose to store multiple branch targets, branch directions, and the terminating instruction type in each trace cache line [28, 33].

The fill unit by Melvin, et al. places dynamically determined instruction information into the decoded instruction cache [21, 6]. The information consists of decoding hints to reduce the latency of variable-length instruction decoding. The fill unit also places dynamic branch information into the decoded instruction cache, including branch targets for two-way branches and information about the instruction that generates the condition code. J. D. Johnson proposes an expansion cache that is similar to the trace cache, but for a statically scheduled architecture. Each line of the expansion cache contains dynamic information on instruction alignment, branch target addresses, and branch

prediction bits [15].

Friendly, et al. focus on compiler-like optimizations that could be performed dynamically in the fill unit within a trace cache entry [8]. The fill unit determines optimizations that can be performed, then rearranges and/or rewrites the instructions within the trace cache line accordingly. These optimizations include marking register-to-register move instructions, combining immediate values of dependent instructions, combining a dependent add and shift into one instruction, and rearranging instructions to minimize the latency through their operand bypass network. Jacobson and Smith propose similar preprocessing steps for the Trace Processors [13].

Lee, et al. propose a decoupled value prediction for processors with a trace cache [17]. In this work, they perform value predictions using a series of four tables at the time an instruction retires. The fill unit then stores the predicted value and type in a value prediction buffer adjacent to the trace cache, similar to the tinted portion of our trace cache. The goal of this work is to remove value prediction from the critical path in instruction fetch and to reduce the port requirements for the four value prediction tables. Lee and Yew also propose a similar system for use with an instruction cache [18].

Per-instruction profiling has been done for the purpose of post-execution analysis and feedback to a compiler. Conte, et al. propose a profile buffer to allow for dedicated profiling [4]. The goal of the buffer is to improve the accuracy of information used in compiler optimizations while designing a hardware-compiler system. Dean, et al. present *ProfileMe* as a hardware approach to sample instructions and paired instructions as they travel through the out-of-order pipeline [5]. The results are designed to provide off-line feedback for programmers and optimizers. Work from Merten, et al. attempts to identify hot spots in order to support run-time optimizations [22]. This work uses dedicated hardware tables and logic to allow for rapid detection of hot spots with negligible overhead. The authors also envision hot spots being used for run-time binary reoptimization [23].

There exists a large body of research on prediction related to the schemes discussed in this paper, which we refer to throughout the paper. However, due to limitation of space, the focus of this section is on related work that has similarity to our general framework.

6 Conclusion

This paper proposes a Traveling Speculation framework, organized around distributed predictors and a traveling hint, called a tint. The tints are distributed on a per-instruction basis, reducing destructive interference and increasing predictor accuracy and instruction coverage. The in-flight tints allow for high-bandwidth predictions because there is no access to a centralized table. Instead

all necessary execution history travels with the instructions and resides in a trace-based hint storage. The tint storage corresponds one-to-one with the instruction storage of the trace cache. This allows for simple tint fetching and also provides path correlation for the history information within the tints. Additionally, counter fields in the tint can dynamically collect profile-like information at run-time for immediate feedback to the microarchitecture.

We describe two tint examples to illustrate the potential of Traveling Speculations. A value prediction tint, VP-Tint, provides a 9.7% performance improvement over the base model for the SPECint2000 benchmarks, and 4.7% additional performance compared to a table-based stride predictor. The VP-Tint provides extra prediction performance by reducing the misprediction rate from 10.3% to 6.1% versus a two read port stride value predictor. In this case, the number of correct predictions almost triples the amount of correct predictions.

A speculative memory forwarding hint, SMF-Tint, provides 11.8% performance improvement over the base model and 4.5% additional performance compared to a memory renaming scheme. The increased instruction throughput is primarily due to a reduction in the misprediction rate from 24.2% to 11.3%. Further analysis reveals that the natural path correlation due to trace storage of tints has a substantial impact on the performance of the speculative memory forwarding tint.

This work indicates that the Traveling Speculation framework can be a powerful, dynamic, self-tuning mechanism for future implementations of prediction techniques in wide-issue microprocessors. The VP-Tint and SMF-Tint are two potential applications of the framework. Depending on the restrictions and design goals of a particular processor, many other useful tints can be designed. For example, a low-power processor could choose to use the run-time profiling abilities of Traveling Speculations for power management schemes. In fact, any history-based policy - not just predictions - can benefit from the Traveling Speculation framework. Cache replacement policies and cluster scheduling algorithms are additional candidates for tints.

We will continue to analyze more applications of the Traveling Speculation framework. In addition, we are looking at solutions to some of the frameworks weaknesses such as lost tint history on trace cache evictions and vulnerability to stale data. We are also working on supporting multiple classes of tints (tints for branches, tints for loads, tints for floating point, etc) and multiple tints per instruction class. We believe that the Traveling Speculation framework can be a powerful tool for dynamically fine-tuning future highly-speculative, wide-issue microprocessors.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer*

Architecture, pages 248–259, Jun 2000.

[2] B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. In *26th International Symposium on Computer Architecture*, pages 196–207, May 1999.

[3] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12 and UWCSE 93-06-06, Sun Microsystems Laboratories, Incorporated, and the University of Washington, 1993.

[4] T. Conte, K. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *29th International Symposium on Microarchitecture*, pages 36–45, Dec 1996.

[5] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *30th International Symposium on Microarchitecture*, pages 294–303, Jun 1997.

[6] M. Franklin and M. Smotherman. The fill-unit approach to multiple instruction issue. In *27th International Symposium on Microarchitecture*, pages 162–171, Nov 1994.

[7] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *30th International Symposium on Microarchitecture*, pages 24–33, Dec. 1997.

[8] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache processors. In *31st International Symposium on Microarchitecture*, pages 173–181, Dallas, TX, November 1998.

[9] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report 1080, Technion - Israel Institute of Technology, Nov 1996.

[10] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th International Symposium on Computer Architecture*, pages 272–281, June 1998.

[11] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ILP. In *International Conference on Supercomputing*, pages 21–28, Jul 1998.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[13] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *International Symposium of High Performance Computer Architecture*, Jan 1999.

[14] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *33rd International Symposium on Microarchitecture*, Dec 2000.

[15] J. D. Johnson. Expansion caches for superscalar microprocessors. Technical Report CSL-TR-94-630, Stanford University, Palo Alto, CA, Jun 1994.

[16] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *25th International Symposium on Computer Architecture*, pages 155–166, Jun 1998.

[17] S. Lee, Y. Wang, and P. Yew. Decoupled value prediction on trace processors. In *6th International Symposium on High Performance Computer Architecture*, pages 231–240, Jan 2000.

[18] S. Lee and P. Yew. On some implementation issues for value prediction on wide-issue ILP processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 145–156, Oct 2000.

[19] M. H. Lipasti and J. P. Shen. Superspeculative microarchitecture for beyond AD 2000. *Computer*, pages 59–66, Sep 1997.

[20] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Labs, Palo Alto, Calif., Jun 1993.

[21] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *21st International Symposium on Microarchitecture*, pages 60–63, Dec 1988.

[22] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying hot spots to support runtime optimization. In *26th International Symposium on Computer Architecture*, pages 136–147, May 1999.

[23] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. A hardware mechanism for dynamic execution and relayout of program hot spots. In *27th International Symposium on Computer Architecture*, pages 47–58, Jun 2000.

[24] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[25] T. C. Mowry and C. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *30th International Symposium on Microarchitecture*, pages 314–320, Dec 1997.

[26] R. Nair. Dynamic path-based branch correlation. In *28th International Symposium on Microarchitecture*, pages 15–23, Nov 1995.

[27] S. T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct 1992.

[28] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace catch fetch mechanism. Technical report, University of Michigan, 1997.

[29] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, 1994.

[30] M. Postiff, D. Greene, and T. Mudge. The store-load address table and speculative register promotion. In *33rd International Symposium on Microarchitecture*, pages 235–244, Dec 2000.

[31] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying load and store instructions for memory renaming. In *International Conference on Supercomputing*, pages 399–407, June 1999.

[32] G. Reinman and N. Jouppi. An integrated cache timing and power model, 1999. COMPAQ Western Research Lab.

[33] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[34] B. Rychlik, J. Faistl, B. Krug, and J. P.Shen. Efficacy and performance impact of value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct 1998.

[35] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, Dec 1997.

[36] Semiconductor Industry Association. The national technology roadmap for semiconductors, 1999.

[37] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The Agree Predictor: A mechanism for reducing negative branch history interference. In *24th International Symposium on Computer Architecture*, pages 284–291, June 1997.

[38] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. *http://www.spec.org/osg/cpu2000/*.

[39] G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communications through renaming. In *30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[40] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th International Symposium on Microarchitecture*, pages 281–290, Dec 1997.

[41] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th International Symposium on Computer Architecture*, pages 124–134, May 1992.