Selective Eager Execution on the PolyPath Architecture*

Artur Klauser, Abhijit Paithankar, Dirk Grunwald
[klauser,pvega,grunwald]@cs.colorado.edu

University of Colorado, Department of Computer Science
Boulder, CO 80309-0430

Abstract

Control-flow misprediction penalties are a major impediment to high performance in wide-issue superscalar processors. In this paper we present Selective Eager Execution *(SEE), an execution model to overcome mis-speculation penalties by executing both paths after diffident branches. We present the micro-architecture of the* PolyPath *processor, which is an extension of an aggressive superscalar, out-of-order architecture. The PolyPath architecture uses a novel instruction tagging and register renaming mechanism to execute instructions from multiple paths simultaneously in the same processor pipeline, while retaining maximum resource availability for single-path code sequences.*

Results of our execution-driven, pipeline-level simulations show that SEE can improve performance by as much as 36% for the go *benchmark, and an average of 14% on SPECint95, when compared to a normal superscalar, out-of-order, speculative execution, monopath processor. Moreover, our architectural model is both elegant and practical to implement, using a small amount of additional state and control logic.*

1 Introduction

Today's high-performance processors are plagued by performance limitations due to branch misprediction recovery cycles. To achieve high performance in super-scalar out-of-order execution processors, it is necessary to predict the outcome of conditional branches in the early stages of the pipeline. Branch prediction redirects instruction fetch to the predicted branch target address, ideally without any interruption in the instruction fetch stream. After a branch misprediction, the instruction stream has diverted from the correct execution path of the program. It takes many cycles before the mispredicted branch executes, at which point the misprediction is discovered and misprediction recovery is initiated. The time from the first mispredicted instruction entering the pipeline to the first correct instruction entering the pipeline after misprediction recovery is commonly referred to as branch misprediction latency. The average misprediction latency is the sum of the architected latency (pipeline depth) and a variable latency, which depends on data dependencies of the branch instruction. The overall cycles lost due to branch mispredictions are the product of the total number of mispredictions incurred during program execution and the average misprediction latency. Thus, reduction of either component helps decrease performance loss due to control mis-speculation.

In this paper we propose *Selective Eager Execution* (SEE) and the PolyPath architecture model, which help to overcome branch misprediction latency. SEE recognizes the fact that some branches are predicted more accurately than others, where it draws from research in branch confidence estimation [4, 6]. SEE behaves like a normal monopath speculative execution architecture for highly predictable branches; it predicts the most likely successor path of a branch, and evaluates instructions only along this path. This single path has access to all machine resources, i.e. execution units and fetch bandwidth, for maximal performance. On the other hand, if the branch prediction is diffident, SEE creates a divergence point and starts fetching and processing instructions from both successor paths after the conditional branch. At branch resolution, instructions along the incorrect path are killed, and processing along the correct path continues without interruption. This eliminates branch misprediction recovery time for the set of branches that have a high likelihood of being mispredicted.

SEE can execute instructions from multiple paths simultaneously, which is similar to a multi-threaded architecture executing instructions from multiple threads at the same time. However, instructions from different paths are more closely related by register data flow between these instructions. SEE tracks instructions from multiple paths using a novel instruction tagging and register renaming scheme described in Section 3.

SEE trades off the reduction in branch misprediction latency with an increase in processing bandwidth requirement. We believe that it will be easier in future architectures to provide higher execution bandwidth, rather than being able to hide increased latency. Current research in other areas, such as dependence and value prediction [11], makes the same underlying assumption and tries to hide data-dependence latencies through increased execution bandwidth.

The rest of this paper is organized as follows. Section 2 describes related work and Sec. 3 introduces the details of the SEE concept. In Sec. 4 we present the simulation environment, which is followed by the discussion of our results in Sec. 5. Finally in Sec. 6 we give some concluding remarks.

*We would like to thank Digital Equipment Corporation for an equipment grant that provide the simulation cycles, support from Hewlett-Packard, and the anonymous referees for providing helpful comments. This work was partially supported by NSF grants No. CCR-9401689, No. MIP-9706286 and in part by ARPA contract ARMY DABT63-94-C-0029.

ISCA '98 Barcelona, Spain

2 Related Work

There are two major categories of work that are concerned with reducing the adverse effects of control flow on processor performance. Work in the first category tackles the control flow problem by reducing the number of mispredictions. There are numerous studies on branch prediction, for example Yeh and Patt [23], McFarling [12], Sprangle [18], and Lee *et al* [9] to name just a few. Multi-block ahead prediction [16] and hierarchical prediction in Multiscalar [7, 14] has been proposed to predict across multiple basic blocks in one cycle.

The second category of work, which is closer related to SEE, strives to reduce the cost of each misprediction. Branch delay slots have been used in some architectures, e.g. MIPS [22], to reduce the cost of misprediction recovery by pushing the branch instruction further into the pipeline before the control flow is changed. Branch delay slots do not provide much advantage in super-scalar, deeply pipelined architectures, however, and contribute to complex processor implementations.

In the Y-pipe architecture [8], the first two stages of a 5-stage pipeline are duplicated. After a branch is fetched, the first two pipeline stages, fetch and decode, are filled with instructions from both sides after the branch. In the third cycle, the branch has completed execution and only the instructions from the correct path continue to flow through the rest of the pipeline. This eliminates the branch misprediction penalty in this simple pipeline altogether.

The work on the misprediction recovery cache (MRC) [1] inserts a small predecoded instruction cache before the execution stage into the middle of a super-scalar, in-order pipeline. This cache holds short instruction sequences following previous mispredictions. After a misprediction, the MRC is searched for a valid entry. If one is found, it is used to feed instructions into the execution stages while instruction fetch is restarted at the correct successor address. The paper evaluates MRC in the context of an in-order CISC pipeline without register renaming.

Disjoint eager execution [21] uses a form of eager execution to speed up the execution of loops. The architecture uses a fixed-size static instruction window. Iterations are mapped to execution paths depending on a predetermined average global branch prediction accuracy, which determines the exact structure of the hardware. Given the fixed structure of the static instruction window, the architecture cannot handle loops larger than the instruction window at its optimal speed. Also, non-looping, control intensive programs cannot make optimal use of the eager execution hardware. The authors estimate a cost of approximately 100 million transistors for a reasonable implementation of the architecture.

Adaptive Branch Tree (ABT) [2] has been proposed to steer execution in a multi-path processor. ABT uses static branch (taken) probabilities to decide which path to execute next. Our work uses similar concepts for tagging instructions from multiple paths, but we dynamically estimate prediction accuracy of dynamic branch predictions to make the path traversal decisions.

Additionally we draw from work in the area of branch confidence estimation [4, 6, 20] to determine if branch prediction or selective eager execution is more effective for each particular occurrence of a branch.

3 The PolyPath Architecture

In the following sections, we first outline the operation of a normal (monopath) speculative architecture, which is commonly found in today's high performance processors. We then discuss the necessary changes in the micro-architecture to implement SEE. We call the resulting architecture *PolyPath*.

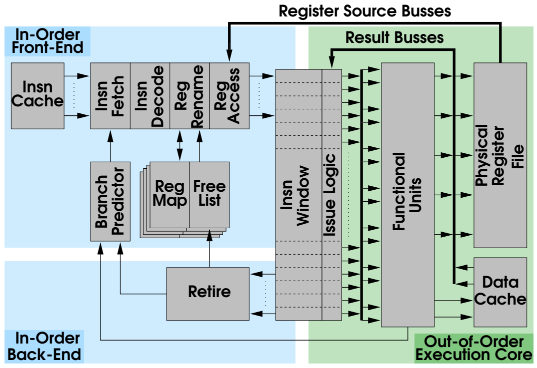

Figure 1. Super-scalar out-of-order architecture model.

3.1 Monopath – Speculative Execution

Figure 1 depicts the micro-architecture of a speculative, super-scalar, out-of-order execution, in-order commit processor. Variations of this architecture model are found in almost all current high-performance microprocessors. We will use an implementation of this architecture model throughout our discussion of performance results in Sec. 5. We will evaluate the performance of our PolyPath architecture by comparing it against the performance of this monopath model. We call this a *monopath* architecture since it always executes only along *one* (speculative) linear sequence of instructions or *path*.

The architecture consists of an in-order front-end and back-end and an out-of-order execution core. Instructions enter the pipeline in program order and stay in-order until they reach the instruction window. The instruction window acts as a buffer between the in-order and out-of-order domains of the architecture. The out-of-order execution core takes the instructions out of the instruction window in data-flow order, computes new results, and puts results into the physical register file. Results are also forwarded to the instruction window and trigger the execution of further instructions. Instructions leave the instruction window in program order. The in-order back-end retires the oldest completed instructions from the instruction window. In this model the central instruction window also acts as a reorder buffer. Some implementations split this functionality into a separate reorder buffer and out-of-order reservation stations to increase circuit speed [22].

The instruction fetch stage uses a branch predictor to predict the fetch address after branches, before the correct address is computed by the back-end. When the branch is resolved in the execution stage, the prediction is compared against the correct outcome. If the prediction was correct, no action is taken. If the prediction was incorrect, all instructions fetched after the branch are discarded. This involves all instructions in the front-end as well as all newer instructions in the instruction window. Instruction fetch is restarted at the correct successor address after the branch. In either case, the branch predictor is updated with the correct branch outcome. Some implementations, such as Pentium Pro, restrict branch resolution to be in-order, i.e. to happen at instruction retirement rather than at instruction completion. With this simplification, all instructions left in the instruction window can be killed at this point, which can result in a simpler circuit implementation.

The architecture uses register renaming to eliminate false dependencies in register data-flow. Instructions encode logical register numbers. The micro-architecture uses physical registers to store results. In the register rename stage each logical register used in an instruction is translated to a physical register number with the help of the register mapping table (RegMap). If the instruction produces a register result, a free physical register is assigned to the logical destination register. The new physical register is assigned from the free list, and both the free list and the register map are updated. The decoded instruction carries the old and the new physical register name of its destination register. The old physical register is needed at commit and rollback after exceptions.

When a branch is in the register rename stage, a checkpoint of the current contents of the RegMap is made. If the branch was mispredicted, the checkpoint RegMap is restored to the current RegMap in order to restore the micro-architecture to the correct state. If the branch was predicted correctly, the checkpoint is destroyed. Checkpoint-restore allows fast recovery from mispredicted branches. Note that the number of available checkpoint RegMaps limits the number of pending unresolved branches in the architecture.

When an instruction retires, its old physical destination register is recycled and put into the free list. At this point, no more instructions can possibly use the old destination register, so it can be reused. Up to this point, there exists the possibility that restoring a checkpointed RegMap that maps a logical register to the old physical register can revive this physical register.

This architecture uses *precise exceptions* [17]. Exceptions, which occur during the execution of an instruction in the out-of-order core, are stored together with the result of the instruction. They are not signaled until an instruction is ready to retire. This allows (1) the exact register state to be restored for the exception handler and (2) prevents instructions on mis-speculated paths from signaling spurious exceptions.

In the monopath model, the following observations can be made:

1. If a branch is predicted correctly, no performance penalty is paid for the control speculation. Also, the predicted (and correct) path has access to all available machine resources, such as functional units and fetch bandwidth.

2. If a branch is mispredicted, all cycles between the branch prediction and misprediction recovery are lost, i.e. they are spent on fetching wrong instructions which do not contribute to the final result. Our measurements indicate that, on average, there are 1.86 times as many instructions fetched as committed, which means 46% of the fetch cycles are wasted.

3.2 PolyPath – Selective Eager Execution

Based on the observations mentioned above, we propose the PolyPath architecture model with the following goals:

1. For correctly predicted branches, no performance penalty should be paid and all resources should be available to the (single) successor path.

2. For incorrectly predicted branches, we strive to reduce the number of lost cycles by introducing some useful work between the misprediction and its discovery. In this way only a fraction of these cycles are lost.

Selective eager execution is based on assessing branch prediction accuracy, which we call branch confidence estimation. Confidence estimation determines if a given branch prediction is likely to be correct or incorrect. If the confidence estimator signals high confidence, SEE follows the branch prediction like a monopath architecture. This is done to achieve goal (1), maximum performance, for predictable branches. If the confidence estimator signals low confidence, SEE diverges and starts processing instructions from

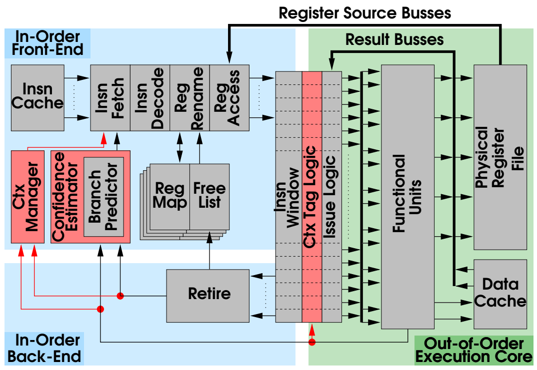

Figure 2. PolyPath architecture model.

both successor paths after the branch. This is done to achieve goal (2), introducing a mix of correct and incorrect instructions after a likely mispredicted path. Multiple divergence decisions can be made before the first divergent branch gets resolved. We call this a *PolyPath* architecture, because it can execute along *many* different instruction sequences, or *paths*, at the same time.

Incorrectly predicted branches with high confidence are handled no differently on the PolyPath architecture than on a monopath architecture; only the predicted (incorrect) path is followed and eventually the misprediction is discovered and the full misprediction penalty is paid. Correctly predicted branches with low confidence are handled less efficiently in the PolyPath architecture, since the correct path has to share fetch bandwidth and execution resources with the incorrect path. Thus, it is important for the performance of SEE to minimize this case and maximize the case of incorrectly predicted branches with low confidence.

To support the operation of SEE in the PolyPath microarchitecture, several changes are necessary to the monopath architecture model. Figure 2 shows an overview of the PolyPath microarchitecture. Two major differences to the monopath architecture exist; (1) a confidence estimator is added to the branch predictor and (2) context tags are added throughout the pipeline. Context tags are described in Sec. 3.2.1 and the confidence estimator is described in Sec. 3.2.7.

3.2.1 Context Tags

In the PolyPath architecture, instructions from many different paths are present at the same time. On misprediction recovery, only the instructions that are in the mispredicted path, or descendants of it, need to be invalidated. In a monopath architecture this is handled by killing instructions depending on their position in the architecture, i.e. all instructions "after" the branch in the instruction window as well as all instructions in the in-order front-end are discarded. In the PolyPath architecture, position alone cannot be used for this purpose, since instructions from the correct path might have been fetched after a divergent branch and should not be killed when this branch resolves.

Context (CTX) tags are used to differentiate instructions from different paths. A CTX tag encodes the branch history that leads to this path, as shown in Fig. 3. The branch history includes the branches between the oldest branch in the pipeline and the current point, such that each branch in this path takes up one history position in the CTX tag. CTX tags are unique for all paths that are present in the pipeline at the same time. However, CTX tags are being reused as old paths leave the pipeline and vacate their corresponding history position. We use CTX tags with 2 bits per history

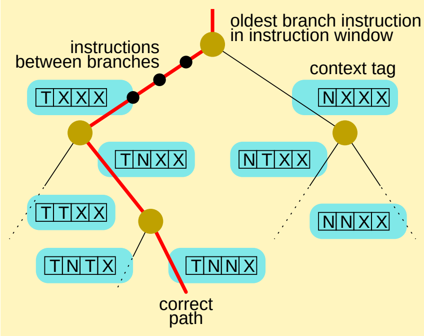

Figure 3. Assigning context tags to execution paths.

History Encoding			
valid	direction	symbol	description
0	0	X	invalid
0	1	X	invalid
1	0	N	not taken
1	1	T	taken
2-bit encoded history position			

Figure 4. Encoding of branch history in context tags.

position as shown in Fig. 4. The first bit encodes the branch direction, and the second bit is a valid bit. If the valid bit is set, the direction bit indicates the appropriate branch direction; otherwise this history position is invalid. A history position can be invalid because the tag was created with a tag length that was smaller than the register that stores the tag. A history position may also be invalid if the branch that occupied this history position has already committed.

We denote the three states of each history position with T, N, and X for "taken", "not taken", and "don't care" (invalid) respectively, as shown in Fig. 4. We always show 4-wide (8-bit) histories to simplify our examples, but this is not a limitation of the actual design. CTX tags are used to define an inheritance relationship between paths in the pipeline, which is why we need the X indication for history positions. The oldest branch in the pipeline is always denoted by a CTX tag of XXXX. Its immediate descendant paths are TXXX and NXXX, which in turn can have descendant paths TTXX, TNXX, NTXX, and NNXX and so on. Not all of these paths might actually be created during an execution sequence. With this tree-structured CTX tag encoding it is simple to find out if two paths are related, i.e. if one of them is a descendant of the other. If we ignore the X (invalid) history positions and find that one tag is a prefix of the other tag, then the longer tag is a descendant of the shorter one.

For example, if we compare tags T(XXX) and TNT(X), we can see that TNT(X) is a second level descendant of T(XXX); comparing TT(XX) with TNT(X), however, shows that they are not related. This property still holds even if we rotate each tag field two positions to the right; *e.g.*, (XX)T(X) and T(X)TN are still considered related. The comparison is independent of the position of branch histories in the tag. This allows the architecture to simply wrap around and reuse vacated history positions without having to re-align CTX tags. New history positions are assigned left to right in the CTX tag. After all history positions have been used, the assignment of new history positions wraps around to the left side of the tag and reuses history positions as they are vacated by commiting branches.

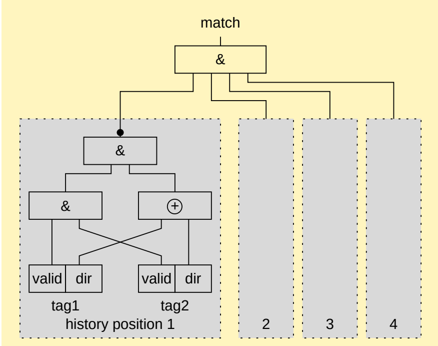

Figure 5. Context tag hierarchy comparator logic.

One possible gate-level implementation of a hierarchy comparator is depicted in Fig. 5. This comparator structure comprises the core functionality of the control-flow state machine logic as described later in Sec. 3.2.3, the store buffer forwarding logic (Sec. 3.2.4), and the CTX table manager (Sec. 3.2.6).

The work on *Adaptive Branch Trees* (ABT) [2] uses a similar scheme to encode the tree-structured relationship of branch paths. ABT uses a 1-bit encoding scheme, which is more restrictive since it only allows in order branch resolution in the tree. All branch tags in this architecture must be re-aligned – shifted by one bit position – when the oldest branch resolves. Our 2-bit scheme is general enough to handle out-of-order branch resolution and does not need tag re-alignment.

CTX tags are stored in fixed length fields, which limits the number of concurrent paths by the bit-width of these fields. Each instruction is marked with a CTX tag in the decode stage. The CTX tag is part of the decoded information stored in the instruction window, together with logical and physical register designators, operation codes, and other microarchitectural control information.

3.2.2 Context Tag Management

The front-end and back-end of the machine are responsible for managing the flow of CTX tags through the microarchitecture. Managing this flow is similar to managing the flow of physical register designators. Whereas physical registers are used to manage data-flow, CTX tags are used to manage control-flow. Physical registers are "created" when an instruction creates a new result. CTX tags are "created" when an instruction creates a new path.

When a divergent branch is decoded, two new CTX tags are created, one for the taken path and one for the not taken path. Each succeeding instruction from the taken or not taken path is then marked with the CTX tag of the respective path. The new CTX tags are generated by appending the branch direction, T or N, to the CTX tag of the divergent branch. The branch itself stores its old CTX tag and the index of its own history position in the new tags. The history position is used for branch resolution, as we will see shortly.

At branch resolution time, the resolved branch CTX tag, branch outcome and its history position are transmitted on the branch resolution bus into the instruction window (see Fig. 6), where all instructions from the wrong path and its descendants are killed. When the branch commits, it is guaranteed that all incorrect successor instructions have been killed and only instructions from the correct path and its descendants remain in the architecture. This also means that all remaining instructions have the same correct history state in the history position of the committing branch. Since this history position is of no more use, it can be invalidated in each

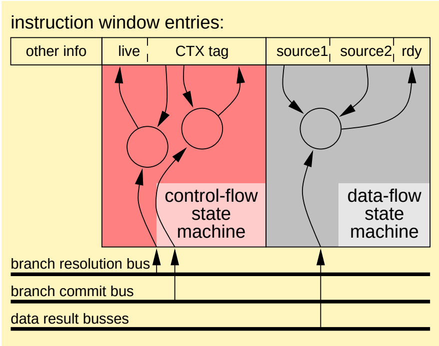

Figure 6. Instruction window logic.

CTX tag. This is simply a matter of resetting one valid-bit in each CTX tag in the instruction window. At the same time, the CTX manager reclaims the history position of the committing branch. This history position can now be reused for new branches, which completes the cycle of CTX tag flow.

The MIPS R10000 processor uses a similar scheme to handle the allocation and deallocation of reservation station entries throughout the architecture. In the R10000 each instruction is marked with a 4-bit *branch mask*, which is equivalent to the valid bits of our CTX tags. At branch misprediction recovery, the branch mask is used to kill in-flight instructions if they are control dependent on the branch. Other instructions in the reservation stations are not affected. When branches resolve correctly, all branch masks in the pipeline reset the one bit corresponding to the resolved branch since this bit can be reused for new instructions.

3.2.3 Instruction Window Context Tag Logic

As we have mentioned above and shown in Fig. 2, each entry in the instruction window stores the CTX tag for its instruction. A small state machine in each instruction window entry snoops branch resolution traffic and matches it with the locally stored CTX tag as shown in Fig. 6. This state machine parallels the logic that snoops result busses and determines when an instruction becomes ready to execute. However, the CTX tag logic is responsible for correct control-flow, rather than data-flow. The following operations are performed by this state machine:

- **store:** The instruction window entry is filled and a new CTX tag needs to be stored locally.
- **resolution:** A branch has resolved and sends its information on the branch resolution bus. The local CTX tag is compared against this CTX tag on the resolution bus. If the local CTX tag is a descendant of the transmitted tag *and* the local tag is on the wrong path, then the state machine signals that this instruction window entry should be killed immediately. Otherwise, no action is taken. If support for multiple branch resolutions per cycle is desired, multiple branch resolution busses are necessary (similar to data-path).
- **commit:** A branch commits and sends its history index on the branch commit bus. The local CTX tag invalidates the history entry corresponding to the transmitted index. This is necessary for the history index to be reused for new branches. Only one commit bus is necessary to support multiple branch retirements per cycle, if the bus carries a decoded valid mask.
- **clear:** When this entry itself commits, its CTX tag is cleared by setting all history fields to invalid.

We are confident that the possibly more complex implementation of the control-flow state-machines, as compared to the data-flow state machines, will not be in the critical timing path, since its inputs arrive earlier in the clock cycle. Current implementations of high performance processors [3] have shown that the result of condition evaluations (branch resolution) is available much earlier in the clock cycle than results from other ALU operations. The Alpha AXP-21164 processor, for example, uses this fact to allow forwarding of condition evaluation results to dependent instructions in the same cycle, resulting in a 0-cycle latency for condition evaluation operations. Also, note that this operation is not in the critical path of instruction execution. An additional cycle latency here has only the effect that wrong-path instructions are killed one cycle later.

3.2.4 Store Buffer

Both monopath and PolyPath architecture models use a store buffer to hold speculative store data before the producing instruction is committed and the result is passed on to the data cache. The store buffer is also responsible for forwarding data to dependent load instructions in the instruction window. In the PolyPath architecture, forwarding needs to be restricted to load instructions in the same path, or a descendant path, of the store instruction. This is achieved by adding CTX tags to store buffer entries. These CTX tags determine the relationship between store and load instructions and whether forwarding between these instructions is allowed. The operation to determine path relationships uses hierarchy comparators as outlined in Sec. 3.2.1 and Fig. 5.

3.2.5 Register Mapping Tables

As described in Sec. 3.1, the active RegMap is checkpointed when a branch is decoded to allow instant recovery of the RegMap if a misprediction is discovered. In the PolyPath architecture this concept is still used, but extended in the following way. For a normal coherent branch with only one successor path, only the normal checkpointing is performed. For a divergent branch, the same operation is performed, i.e. a RegMap copy is created. However, since the branch cannot be mispredicted – both successor paths are followed – we use one copy of the RegMap for each successor path. Note that the PolyPath architecture uses only two RegMap copies for every type of branch, the same number of RegMaps that a monopath architecture needs. The difference is that PolyPath uses one RegMap copy for each successor path of a divergent branch, whereas a monopath architecture uses one copy for the predicted successor path and keeps a backup copy in case the prediction was wrong and the other path needs to be executed later. The same CTX tags that are used for marking instructions are also used for tagging RegMaps. Each logical to physical register translation now also requires the CTX tag of the instruction requesting the translation in order to index into the correct RegMap for this CTX .

We believe that his operation will be the most time critical in our PolyPath design, since the additional indexing must fit into the clock cycle. We are confident that it is possible to build such a RegMap with the same timing performance as the original RegMap for a monopath architecture. One promising direction for implementing the RegMap is the use of a 3D register file [19]. However, in Sec. 5.3.4 we also provide results for the case where an additional cycle for this operation might become necessary.

3.2.6 Context Management

The front-end also performs CTX management operations. This involves CTX tag management, as described in Sec. 3.2.2, as well as CTX fetch path management.

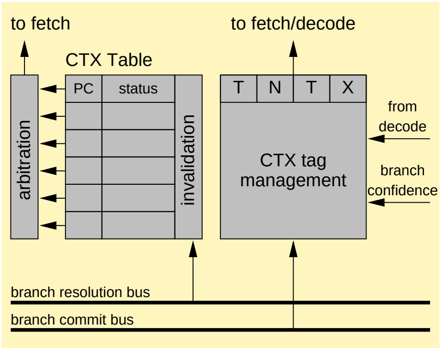

Figure 7. Context manager.

The CTX manager contains a CTX table with one entry per possible CTX in the implementation. The number of possible contexts is limited by the bit-width of the CTX tag field in instructions throughout the architecture. Each entry contains the current fetch PC and the status of the CTX as shown in Fig. 7. The CTX status consists of information such as fetch status (active/stopped), liveness status (live/dead), branch resolution status, etc.

All live, actively fetching paths in the CTX table contend for fetch bandwidth in the fetch stage. Fetch arbitration can have a considerable impact on PolyPath performance. Investigation of fetch policies is a topic of future work. However, we keep the same limit on the aggregate fetch bandwidth for both monopath and PolyPath architecture models in our performance evaluation.

3.2.7 Confidence Estimator

The PolyPath architecture uses a confidence estimator to assess the quality of branch predictions and guide SEE decisions during the execution of a program. If the confidence estimator signals low confidence in a branch decision, the PolyPath architecture processes a branch divergence; otherwise the branch is executed in the normal (monopath) way. Another possible aspect of the confidence estimator is the guidance of fetch bandwidth arbitration. However, we do not explore this dimension in this paper.

To show the effectiveness of the PolyPath model, we use a dynamic branch confidence estimator as described by Jacobsen *et al* [6]. We believe that we can enhance the performance of SEE by better matching the confidence estimator to our application. For this matter, we are looking at applying techniques as discussed in [4] to SEE. An in-depth discussion of alternative confidence estimators, however, is outside the scope of this paper.

This concludes the overview of our PolyPath architecture. In the following sections we will present a performance evaluation of the PolyPath architectural model and compare it to the performance of a monopath architecture.

4 Simulation Environment

4.1 Benchmarks

In this paper, we present results for the eight SPECint95 benchmarks. The benchmarks are compiled with DEC GEM-CC with the highest optimization level (-migrate -O5 -ifo). We use scaled down input data sets for some benchmarks to reduce simulation time. All benchmarks are simulated to completion. Table 1 lists the characteristics of these benchmarks on our architecture.

Benchmark	Instructions (millions)	Branch misprediction
compress	113.8	9.13%
gcc	334.1	11.09%
perl	249.1	8.27%
go	549.1	24.80%
m88ksim	552.7	4.20%
xlisp	216.1	5.20%
vortex	234.4	1.85%
jpeg	347.0	8.37%
average	324.5	7.17%

Table 1. SPECint95 benchmark characteristics

4.2 Architecture Model

We used an extended version of the AINT architecture simulation tool [13] to build a micro-architecture level simulator for the PolyPath architecture as depicted in Fig. 2 and described in Sec. 3.2. Our AINT simulator is execution-driven and handles native DigitalUnix executables with the Alpha instruction set. The simulator performs a detailed cycle-level simulation of the PolyPath architecture and faithfully executes all instructions from both paths after divergent branches until the branch resolves and one path is killed. All internal contention of the instruction window and functional units are modeled exactly as a real implementation of the architecture would experience it.

Our simulated baseline machine model is an 8-way superscalar, out-of-order execution, in-order commit architecture, with a 256 entry central instruction window/reorder buffer and an 8-stage pipeline. Instruction latencies and functional unit compositions are taken from the DEC Alpha AXP-21164 processor [15], and reflect a 500 MHz execution core. All simple integer instructions have a latency of one cycle. More complex instructions, like integer multiply and all floating point operations, have higher latencies as specified in the AXP-21164 hardware reference manual.

The baseline model has 8 integer execution units (4 IntType0 ALUs, 4 IntType1 ALUs), 8 floating point execution units (4 FPAdd, 4 FPMult), and 4 Dcache memory ports. Although this might seem excessive by current standards, we believe that advances in integrated circuits will make this setup feasible in the near future. We also provide scalability results for different instruction window sizes and functional unit configurations in Sections 5.3.2 and 5.3.3. Additionally, we make the following assumptions:

- The instruction and data caches are non-blocking and have an access time of 1 cycle, which results in a 2-cycle load latency (address computation + cache access). Accesses to both caches always hit in the cache.
- The instruction fetch mechanism allows accesses to multiple basic blocks in the same cycle. Accesses from multiple paths are prioritized based on the position of the paths with respect to the oldest uncommitted branch. Fetch bandwidth decreases exponentially with the distance of a path from the oldest branch.
- Perfect memory disambiguation is used. Loads dependent on previous stores can only execute after the store. The architecture forwards store results to dependent loads in the instruction window in one cycle.
- We use a gshare branch predictor as described by McFarling [12], which uses a global history *xor* branch address to index into a table of saturating 2-bit counters. We use a 14-bit global history and 2^{14} (16 k) counters for the baseline architecture. Section 5.3.1 provides some scalability results on the

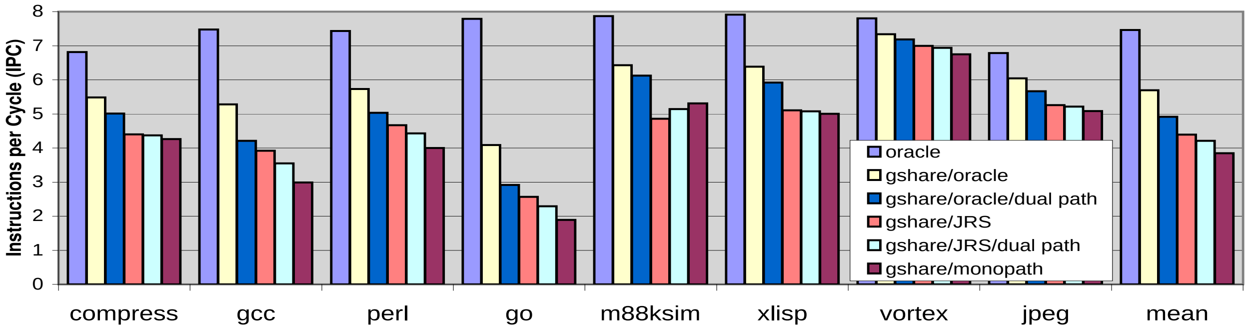

Figure 8. Baseline architecture performance.

influence of the branch predictor size on SEE performance. The global history is speculatively updated at branch prediction with the predicted branch outcome. The correct history state is restored at misprediction recovery time. Speculative history update improved the overall branch prediction accuracy by approximately 1% in our simulations.

- We use a modified version of the Jacobsen-Rotenberg-Smith (JRS) 1-level branch confidence estimator with resetting counters as described in [6]. The indexing method into the predictor is the same as for gshare. However, each counter in the confidence estimator reflects the number of correct predictions since the last branch misprediction for this index. We use the same number of counters in the branch confidence estimator and in the branch predictor in all cases, e.g. 16 k for the baseline architecture. Rather than the 4-bit counters advocated by Jacobsen *et al*, we found that 1-bit counters result in the best performance for our application. The JRS estimator with 1-bit counters achieves much higher PVN[1] than the 4-bit counter version, which means that a higher fraction of the "low-confidence" branches are actually mispredictions. PVN is the most important design parameter for our application of branch confidence estimators.

  We have enhanced the confidence estimator indexing method to use the speculative outcome of the current branch as part of the global history when indexing into the branch confidence estimator counter table. This resulted in a substantial performance improvement over the original indexing method, which uses the same global history as used to index into the branch predictor table [6].

In each cycle our baseline architecture can:

- fetch up to 8 instructions from one or multiple paths;
- decode and rename up to 8 instructions that have been fetched in previous cycles;
- issue one instruction to each functional unit, dependent on the availability of (1) data-ready instructions in the instruction window and (2) free functional units;
- write back one result into the instruction window/reorder buffer from each functional unit that produces a result in this cycle;
- commit up to 8 of the oldest instructions from the reorder buffer if they are already completed;

[1]PVN: Predictive Value of a Negative Test; the probability that a "low-confidence" estimation is for a mispredicted branch

5 Results and Discussion

5.1 Baseline Performance

We show the performance for SEE by first giving bounds on the potential performance improvements that are attainable by better branch prediction. Figure 8 shows the performance of all benchmarks on our baseline machine architecture with normal speculative execution (monopath). In the same figure, the performance of the same architectural model with a perfect branch predictor is depicted as "oracle". The difference between these two categories shows the performance lost due to branch mispredictions. The (harmonic) mean group shows that performance could be improved by approximately 94% by reducing the number of cycles spent due to branch mispredictions. For individual benchmarks, the possible performance improvement is mainly a function of the branch prediction accuracy for this benchmark. For example, the *go* benchmark, experiencing a misprediction rate of 25%, shows the largest potential improvement of more than 300%.

Next, we introduce a perfect branch confidence estimator on top of the gshare branch predictor (gshare/oracle). Although not practically realizable, the perfect branch confidence estimator provides a calibration point for the mere concept of SEE without taking into account the imperfections of specific real confidence estimators. Figure 8 shows that, on average, SEE is capable of making up one half of the performance lost due to branch mispredictions. Again, the performance potential is mainly a function of the accuracy of the branch predictor.

Finally, the category "gshare/JRS" shows the performance of SEE when we use a real branch confidence estimator. We see that we can achieve performance improvements for all but one benchmark (m88ksim), with an overall average of 14% improvement over monopath or an average increase of 0.55 in IPC. This improvement is a function of both the accuracy of the branch predictor as well as the accuracy of the branch confidence estimator. Note that we use a branch confidence estimator that has been published elsewhere in the literature [6], and have not particularly tuned this estimator for our application. We believe that higher speedups are attainable with realistic confidence estimator implementations.

As we see in Fig. 8, SEE can even improve performance for the *vortex* benchmark, which has a misprediction rate of only 1.85%. SEE achieves 4% improvement in this case. We believe that this is also indicative for the potential to obtain performance improvements on other highly predictable programs, like floating point code.

For *m88ksim*, the SEE architecture fails to achieve performance improvements and falls off 8.5% from the baseline monopath performance. Analyzing this benchmark in detail reveals that the Jacobsen-Rotenberg-Smith (JRS) confidence estimator only achieves a PVN of 16%, i.e. only 16% of the "low confidence" decisions are mispredictions and 84% are correct predictions. The average PVN for all other benchmarks is above 40%. A low PVN means that the confidence estimator often errs on the side of signaling low confidence for a branch which is actually predicted correctly. In this case SEE looses performance compared to monopath execution since monopath only executes the correct path, whereas SEE also executes the incorrect path, which leads to resource contention between the two paths. At first, we thought that the low PVN would increase functional unit contention which would lead to the performance loss. However, we found that this is not the case. To the contrary, functional unit utilization is even slightly decreased for SEE. The overloaded resource in this case turns out to be instruction fetch bandwidth. Due to the high fraction of unnecessary divergence decisions, the effective instruction fetch bandwidth for the correct path is diminished to a point where it becomes the primary system bottleneck. This is also reflected in the total number of "useless" instructions that are fetched, i.e. instructions that do not commit. SEE is able to decrease the number of useless instructions in the other benchmarks on average by 15%. Due to the frequent confidence estimator mistakes, SEE increases the number of useless instructions for *m88ksim* by 29%. The lesson we learned from this experience is that a successful branch confidence estimator for SEE should be able to monitor its performance dynamically and revert back to strict monopath execution (always indicating "high-confidence") if it makes too many errors.

5.2 Dual Path Execution

In work reported by Heil and Smith [5] and Lick, Tyson, and Farrens [10, 20] the concept of *dual-path* execution addresses many of the same problems that we address with SEE. Both concepts are only evaluated with trace based simulations. Heil and Smith assume a doubled fetch stage to accommodate the second path and do not model resource contention in the instruction window and functional units due to wrong path instructions. Lick and Tyson do not employ a pipeline-level simulator but only count instructions of executed basic blocks and stop branch fan-out a certain number of branches after the last divergence.

We compare the idea of dual-path execution to our more general notion of SEE by restricting our PolyPath simulator to only one divergence point, 3 paths[2], in the pipeline at any time. Figure 8 shows the performance of oracle dual-path (gshare/oracle/dual-path) and real dual path (gshare/JRS/dual-path) on our architecture model. From this evaluation we see that oracle dual-path can achieve on average only 58% of the performance improvement of oracle SEE. Real dual-path achieves on average 66% of the performance improvement of real SEE (gshare/JRS). Looking at the path utilization for SEE reveals that the average number of active paths is only 2.9. SEE uses 3 paths or fewer approximately 75% of the time, which is why dual-path execution can get a substantial fraction of the performance improvements of SEE. We believe, however, that the more general concept of SEE has similar implementation costs as a realistic dual-path implementation, while at the same time resulting in higher performance improvements.

5.3 Architectural Variations

To validate our results over a range of architectural changes we have simulated SEE on architectures that vary several parameters of our baseline machine model and measured its performance. In the following graphs, we present the performance of oracle branch prediction (oracle), monopath execution (gshare/monopath), and SEE with oracle confidence estimation (gshare/oracle) and with the JRS confidence estimator (gshare/JRS). We show the (harmonic) mean performance across all benchmarks. Whenever we use the unqualified term "SEE" in the following discussion, we are referring to the SEE architecture with JRS confidence estimation.

[2]Dual-path execution is equivalent to using 3 paths in the PolyPath architecture, since PolyPath assigns one path to the instructions before the divergent branch and one path each to the instructions in either direction after the branch.

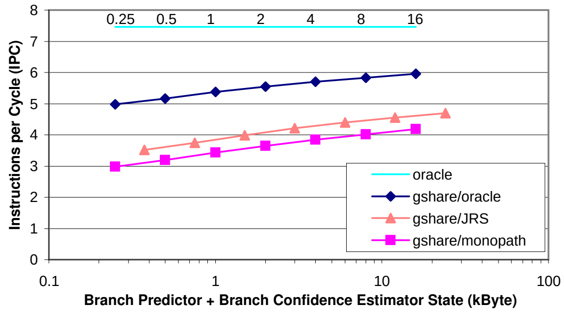

Figure 9. Branch predictor sizes.

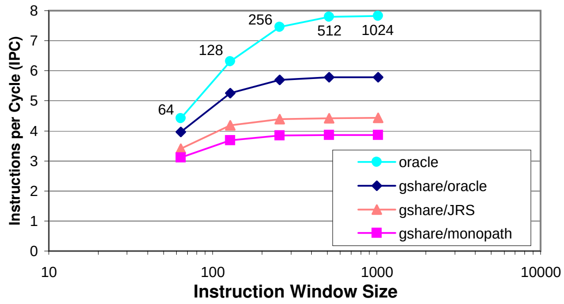

Figure 10. Instruction window sizes.

5.3.1 Branch Predictor Size

In the first set of scalability experiments, we vary the size of the branch predictor. Figure 9 shows the performance for branch predictor sizes of 1k to 64k 2-bit counters (0.25 to 16 kByte state). The geometric mean misprediction rate ranges from 16.3% to 6.4% respectively. To allow an equal-area comparison between different architectures we plot the performance as a function of the number of bytes used for all predictors in the system. The combined branch predictor and JRS confidence estimator is 1.5 fold the size of the branch predictor alone because of the additional table of 1-bit confidence counters. We see that over the entire range of branch predictor sizes, SEE is able to achieve a constant absolute gain of approximately 0.5 IPC over monopath. This corresponds to 15% and 10% relative performance improvement over monopath for the smallest and largest branch predictors respectively.

Another way to look at this data is to follow iso-performance lines. This shows that, for example, a SEE architecture with a 12-bit history scheme (1.5 kB state) can reach a performance of approximately 4 IPC. To reach the same performance the monopath architecture needs a branch predictor with a 15-bit history scheme (8 kB state). Thus, monopath needs to store approximately 5.3 times as much state information as SEE to achieve the same performance.

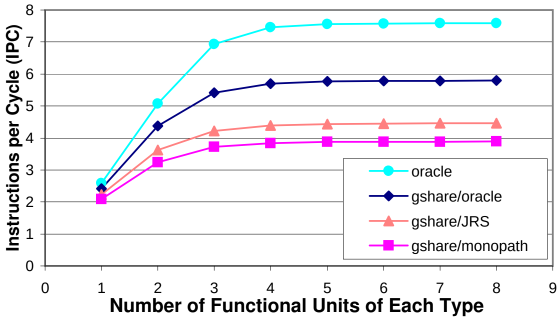

Figure 11. Functional unit configurations.

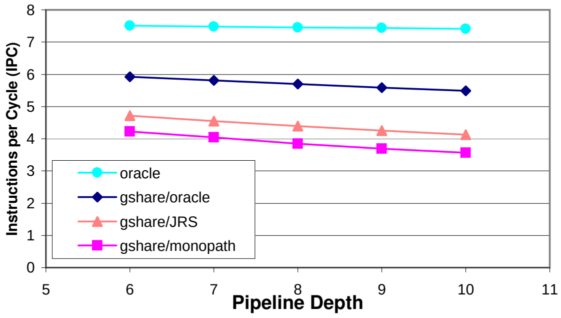

Figure 12. Pipeline Depth.

5.3.2 Instruction Window Size

Our next set of scalability experiments shows the influence of the instruction window size on SEE performance. Figure 10 shows the performance for instruction window sizes ranging from 64 to 1024 entries. The performance with oracle branch prediction shows that instruction window sizes larger than 256 entries only show minimal performance improvements over a 256 entry instruction window. The performance of gshare-based branch prediction schemes reaches most of its full potential even earlier, with a 128 size instruction window. In fact, the average usage of the instruction window saturates at approximately 145 instructions (average over all benchmarks) with the gshare branch predictor. For instruction window sizes less than 256 entries the performance of some benchmarks (compress, jpeg) starts to suffer significantly from the reduced scheduling freedom, and falls off rapidly. However, note that SEE is still able to perform approximately 9% better than monopath for a 64 entry instruction window, which is about 38% of the potential performance improvement for SEE with oracle confidence estimation.

5.3.3 Functional Unit Configuration

Our baseline architecture has a very potent execution core configuration. To test the SEE architecture with less aggressive architecture implementations, we conducted a set of experiments that assesses the influence of the number of functional units on SEE performance. Since SEE uses a more aggressive execution model than monopath, we assumed that it would benefit more from additional execution bandwidth then monopath.

Figure 11 shows the performance as a function of the number of functional units. The x-axis shows the number of functional units of each type as well as the number of memory ports. There are four types of functional units in our execution core, IntType0 ALUs, IntType1 ALUs, FPAdd, and FPMult. For the data points at x=4, we have a total of 8 integer ALUs and 8 floating point ALUs, as well as 4 Dcache memory ports. We have chosen to scale the number of all functional units uniformly for this experiment, although this would probably not be done in a real implementation. In addition, since we only use integer benchmarks in this paper, having more than one FPAdd and FPMult only has negligible effects on the performance of our benchmarks.

We can see from Figure 11 that SEE improves performance over monopath for all functional unit configurations. The relative performance improvement ranges from 14% for 3 and more functional units, down to 6% improvement for 1 functional unit. So, even in the case where the primary bottleneck of the architectural configuration is the number of functional units (1 and 2 in our graph), SEE is able to perform consistently better than monopath execution.

We conclude from this experiment that high bandwidth execution cores help to increase the performance improvement of SEE over monopath but are not imperative to get performance improvements from SEE architectures.

Even the configuration with only 1 functional unit of each type has enough spare capacity in its functional units to benefit from putting more pressure on them. The spare capacity is created by data-dependency stalls. For example, for monopath we see a utilization of IntType0, IntType1, and Dcache port of 81%, 75%, and 75% respectively. For SEE, this increases to 85%, 80%, and 80% respectively. It is this usage of spare functional unit capacity that allows SEE to achieve its overall performance improvement of 6% in this case.

5.3.4 Pipeline Depth

In the final set of experiments, we evaluated the influence of the pipeline depth on the performance of the SEE architecture. We varied the depth of the processor pipeline by changing the number of pipeline stages in the in-order front-end of the architecture (fetch, decode). Figure 12 shows the performance for total pipeline lengths of 6 to 10 stages. As the pipeline gets longer, branch misprediction latency gets higher which causes the overall performance to decrease slowly.

Comparing the performance for equal length pipelines shows that SEE with JRS confidence estimator is able to slightly increase its absolute performance improvement from 0.49 IPC for a 6 stage pipeline to 0.56 IPC for a 10 stage pipeline. This corresponds to a relative performance improvement of 11% and 16% respectively. The higher performance improvement for longer pipelines can be attributed to the fact that the misprediction penalties, which were avoided by SEE, are more costly for the monopath architecture in the longer pipelines.

An alternative way to interpret this data is to look at iso-performance lines. For example, to get a performance of 4 IPC we need to implement a monopath architecture with a short 7 stage pipeline. SEE is able to reach the same performance with a pipeline as long as 10 stages. If the additional pipeline stages are used to reduce the length of the critical timing path, the clock cycle can be shortened. This again results in higher performance, i.e. shorter program runtimes.

In Section 3 we have assumed that the additional complexity of SEE can be included in a pipeline without adverse effects on cycle time or pipeline length. The results in Figure 12 show that SEE achieves performance improvements even if we had to extend the pipeline to accommodate SEE. Comparing the base case of an 8-stage monopath pipeline to an 8, 9, and 10-stage SEE pipeline shows that SEE is able to increase the performance by 14%, 11%, and 7% respectively. Even though performance improvements decrease, a one to two stage extended pipeline for SEE is still an attractive option compared to the shorter monopath pipeline.

6 Concluding Remarks

Processor performance can be severely limited by the penalties incurred due to branch misprediction recovery. This is particularly the case for wide superscalar architectures with deep pipelines. This problem will get worse in the future as machine widths get larger and pipelines get deeper to cope with shorter cycle times and higher complexity. Our model architecture of an 8-way superscalar, 8-stage deep pipeline machine shows a 50% performance loss due to branch misprediction cycles. Two factors contribute to misprediction recovery cycles: number of mispredictions and average misprediction latency. We have presented the Selective Eager Execution (SEE) concept and the PolyPath architecture that strive to reduce misprediction latency by executing down both paths after diffident branch predictions. The PolyPath architecture uses a novel instruction tagging scheme that allows it to handle instructions from many different paths in the pipeline at the same time.

Our simulation results show that SEE has an average performance potential of 48% speedup over monopath speculative execution, which is approximately half of all performance lost due to branch mispredictions. We have shown that a sample SEE architecture with a Jacobsen-Rotenberg-Smith branch confidence estimator is able to achieve an average performance improvement of 14% over monopath, and a maximum improvement of 36% for the *go* benchmark.

We have also shown that the performance improvements of SEE are robust against changes of various parameters of the underlying machine architecture. SEE achieves its performance improvements by harnessing unused functional unit capacity.

Several areas of the PolyPath architecture are only covered in concept in this paper and are subject to future work. In particular, we think that instruction fetch design and arbitration is a critical issue that needs to be investigated in more detail.

References

[1] J. Bondi, A. Nanda, and S. Dutta. Integrating a Misprediction Recovery Cache into a Superscalar Pipeline. In *29th Intl. Conf. on Microarchitecture*, December 1996.

[2] Tien-Fu Chen. Supporting Highly Speculative Execution via Adaptive Branch Trees. In *Fourth Intl. Symp. on High-Performance Computer Architecture*, February 1998.

[3] John Edmondson *et al*. Internal Organization of the Alpha 21164. A 300 MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[4] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence Estimation for Speculation Control. In *25th Intl. Symp. on Computer Architecture*, June 1998.

[5] Timothy Heil and James Smith. Selective Dual Path Execution. Technical report, University of Wisconsin-Madison, November 1996. http://www.ece.wisc.edu/˜jes/papers/sdpe.ps.

[6] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 142–152, Paris, France, December 1996.

[7] Q. Jacobson, S. Bennett, N. Sharama, and J. Smith. Control Flow Speculation in Multiscalar Processors. In *3rd Intl. Symp. on High Perf. Computer Architecture*, pages 218–229, February 1997.

[8] Michael J. Knieser and Christos A. Papachristou. Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays. In *25th Intl. Conf. on Microarchitecture*, pages 125–128, 1992.

[9] Chih-Chieh Lee, I-Cheng Chen, and Trevor Mudge. The Bi-Mode Branch Predictor. In *Proceedings of the 30th Annual Intl. Symp. on Microarchitecture*, Research Triangle Park, North Carolina, December 1997.

[10] Kelsey Lick. Limited Dual Path Execution. Master's thesis, University of California, Riverside, 1996.

[11] Mikko Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[12] Scott McFarling. Combining Branch Predictors. TN 36, DEC-WRL, June 1993.

[13] Abhijit Paithankar. AINT: A Tool for Simulation of Shared-Memory Multiprocessors. Master's thesis, University of Colorado at Boulder, 1996.

[14] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace Processors. In *30th Intl. Conf. on Microarchitecture*, December 1997.

[15] Digital Semiconductor. Alpha AXP-21164 Processor Hardware Reference Manual, September 1997.

[16] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-Block Ahead Branch Predictors. In *Proceedings of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Cambridge, Massachusetts, October 1996.

[17] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[18] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt. The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In *24th Annual Intl. Symp. on Computer Architecture*, pages 284–291, May 1997.

[19] M. Tremblay, B. Joy, and K. Shin. A Three Dimensional Register File for Superscalar Processors. In *28th Hawaii Intl. Conf. on System Sciences*, pages 191–201, January 1995.

[20] Gary Tyson, Kelsey Lick, and Matthew Farrens. Limited Dual Path Execution. CSE-TR 346-97, University of Michigan, 1997.

[21] A. K. Uht, V. Sindagi, and K. Hall. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *28th Intl. Conf. on Microarchitecture*, pages 313–325, December 1995.

[22] Keneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, February 1996.

[23] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *20th Intl. Symp. on Computer Architecture*, pages 257–266, May 1993.