Confidence Based Out-of-Order Renaming for Speculatively Multithreaded Processors*

Kshitiz Malik, Kevin M. Woley, Samuel S. Stone,
Mayank Agarwal, Vikram Dhar, Matthew I. Frank
Electrical and Computer Engineering
University of Illinois, Urbana-Champaign

June 9, 2006

Abstract

Speculatively multithreaded processors find parallelism by speculatively fetching and renaming dynamic flows of instructions from (perhaps) widely seperated parts of the program flow graph. These processors must handle inter-thread register dependences. The approach followed in this paper is to dynamically identify the consumers of interflow register mappings that will be (but have not yet been) produced in a logically earlier thread and then to dynamically awaken those consumers as soon as the mapping they are waiting for is produced.

The main contribution of this paper is the design and evaluation of the inter-thread register renaming and synchronization mechanisms for a speculatively multithreaded processor that does not need compiler support. Our scheme is realizable, aggressive, and flexible and achieves speedups within about 10% of those achievable by an oracle. We find that inter-thread synchronization mechanisms can and must use path confidence information so that the producers of register mappings can awaken consumer instructions at just the right time, neither so early that the producer is on a misspredicted branch path, nor so late as to add latency to the critical path. We also demonstrate that a relatively straight-forward predictor can find the set of consumer instructions that must wait without being overly conservative.

1 Introduction

Superscalar processors are profitable because they issue and execute instructions out of order, but retire instructions in order, thus providing high performance on a programming model that is easy to reason about. However, superscalars fetch and rename instructions in program order and cancel all instructions after a branch mispredict, even when those instructions have done useful work. Speculatively Multithreaded processors [13, 14, 6, 7, 4, 1, 11, 8, 9] are a promising alternative because they retire instructions in order, like a superscalar, but also fetch and rename instructions out of order. This allows them to find instruction level parallelism across widely separated regions of the program, including past multiple branch mispredictions.

Although fetching out-of-order improves fetch efficiency, it introduces complications because of dataflow that crosses thread boundaries. In particular, there may be register and memory value traffic between the in-order instructions that have not yet been fetched and the out-of-order instructions that have been fetched early by the speculative multithreading mechanism. That is, producers and consumers of inter-thread value traffic sometimes need to be *synchronized*. It turns out (empirically) that more often than not in speculatively multithreading systems, a program's value traffic occurs either within a thread or from a producer instruction that fetched, renamed and executed before a particular thread was even spawned [13]. Thus, many of the instructions that are fetched out-of-order can be renamed and executed in the order they are fetched, while others need to wait for portions of the intermediate path to be fetched. And, when inter-thread register communication does need to be synchronized the synchronization is often on the critical path.

This paper investigates how aggressively (speculatively) the processor should synchronize producer threads with the consumer instructions that depend on them. Similar to many earlier speculatively multithreaded systems [7, 1, 8], the system we evaluate, which we call the PolyFlow Speculative Multithreaded Processor is based on a simultaneously multithreaded core, rather than, for example, a chip multiprocessor. This means that in our system we have the option of allowing producer instructions to forward data to consumer instructions *speculatively*, rather than waiting for all the branches before the producer instruction to complete. We find that *producers must forward data speculatively to consumers* to avoid adding latency to critical

*University of Illinois, Center for Reliable and High Performance Computing, Technical Report Number UILU-ENG-06-2208.

paths, but that the *speculative forwarding of data must be balanced with path confidence information* to avoid producers that are along mispredicted branch paths from signaling consumers too early and thus causing the consumers to also be canceled. The PolyFlow system is a *Dynamic* Speculative Multithreaded system, in that it takes an unmodified binary, and converts it dynamically into threads. Thus, our register synchronization mechanism must handle inter-thread register dependence without any help from the compiler.

The contributions of this work include, first, **the design and analysis of an effective and realizable out-of-order register renaming mechanism for dynamic speculative multithreading.** Our scheme supports out-of-order spawning and reconnect of flows, communicates register information only point-to-point from a predecessor to its immediate successor flow (rather than globally), requires no selective reexecution, and yet achieves speedups within about 10% of those achievable with an oracle that "knows" the *optimal* time for producers to awaken consumers.

Second we demonstrate that **out-of-order register renaming requires path confidence information to balance synchronization between register producers and consumers.** We find that register producers must aggressively and speculatively release/awaken consumers to avoid delays waiting for branches to complete and retire. The producers must also, however, take branch confidence information into account to avoid awakening consumers too early, with data from along a mispredicted path. We present the design of an appropriate confidence predictor and show how to use it to drive the register renamer.

Third we demonstrate that **inter-thread register consumers can be identified dynamically**. Our system does not use a compiler to identify consumer instructions, but rather derives this information at runtime. While finding the last dynamic instance of a producer is a difficult problem [7] that requires backward compiler analysis [15], we find that the set of architectural registers that need to be synchronized is highly predictable. Since identifying these register's consumer instructions (and their transitive dependents) is a forward analysis it can be performed at runtime in the front end of the processor. We find that a predictor with 1-bit per architectural register per spawn point is sufficient to allow us to identify the consumer instructions that must wait for a producer instruction from another flow.

The rest of this paper is structured as follows. The next section gives a motivating example to explain, in rough terms, the problem we are trying to solve and our solution, and discusses the relationship of our work to earlier work in speculative multithreading. Section 3 gives a more detailed description of our design. In Section 4 we demonstrate that our aggressive renaming scheme provides speedups within about 10% of that achievable with an oracular confidence predictor driving the time at which producers release consumers. Section 5 concludes.

2 Background

In this section, we describe the register synchronization problem faced by all speculatively multithreaded systems. In our domain of SMT-based speculative multithreading with a shared physical register file, the register synchronization problem boils down to ensuring that all instructions get source physical mappings produced by their corresponding producer instructions, instead of some predecessor of the producer. Hence, we term register synchronization as the problem of performing out-of-order renaming.

A speculatively multithreaded system needs to address three issues regarding inter-thread register synchronization. First the specific instructions that *consume* inter-thread register data need to be identified, and be forced to wait (i.e., their renaming must be delayed) till the corresponding producer instruction has been renamed. We call such instructions *waitsFor* instructions. Second, waitsFor instructions need to be *released*, some time after the producer instruction has been renamed. Release here refers to the process of providing a waitsFor instruction the correct source register mappings. If consumers are released too early (before the correct producer instruction has been renamed) then the thread containing the consumer instruction will need to be canceled. On the other hand if the consumers are released too late the synchronization cost will add to the critical path and slow down program execution. Finally, if consumer instructions are identified speculatively or released speculatively, there needs to be a *validation* process that makes sure that each consumer got matched with the correct producer instruction.

2.1 Example

Figure 1 shows an oversimplified pedagogical example intended to clarify these three issues. In this example thread 0 has spawned thread 1, and then thread 0 has entered a simple if-then-else statement starting at instruction A1. The first issue is to identify waitsFor instructions in thread 1, i.e., instructions which should wait before they go through the rename process. In this case, instruction D1 does not need to wait, because it can immediately retrieve its local register alias table mapping for register Rx (created by instruction Z1, and copied during the spawn process). Likewise instruction D5 does not need to wait, because register Rz has already been renamed locally by instruction D4. Instruction D2, on the other hand, must wait for a mapping to be produced by either instruction B1 or C1. In-

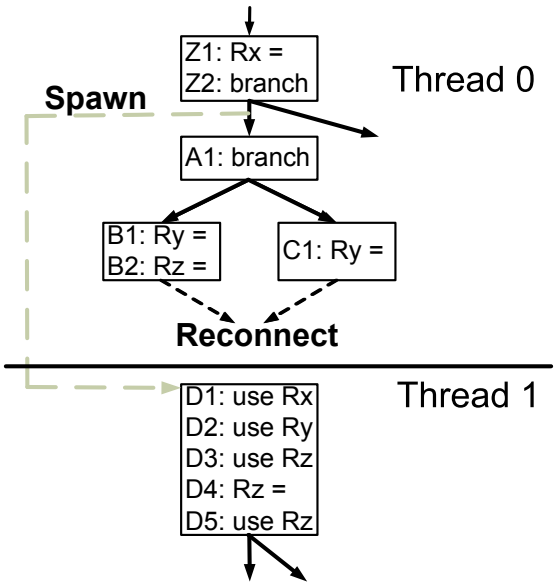

Figure 1: *How aggressively thread 0 should wake up instructions D2 and D3 in thread 1 depends on the predictability of the branch at A1. If the branch is highly predictable, then thread 1's instructions should be awakened as soon as thread 0 predicts through the branch, and reaches the potential reconnection point, marked "reconnect," correctly renaming the intervening instructions in either block B or C. If the branch at A1 is hard to predict then thread 0 should wait until A1 is actually resolved before releasing instructions D2 and D3 in thread 1*

struction D3 may or may not need to wait for the mapping produced by instruction B2, depending on the direction of branch A1. In Section 3 we demonstrate how to use the renamer to derive a non-conservative set of instructions that should wait.

The second question, of when to release the waiting instructions, depends on whether the branch at A1 is highly predictable or not. If the branch is highly predictable then as soon as the renamer *speculatively* renames the predicted block (B or C), instructions D2 and D3 should be released. If, on the other hand, the branch at A1 is not predictable, then it will be better to make D2 and D3 wait until the branch at A1 is resolved so as not to force a flush in thread 1 because of a branch mispredict in thread 0. *This represents a fundamental trade off between the benefit of multithreading systems that branch mispredictions in different threads can be handled* independently *against the cost of adding synchronization to the critical path.* We demonstrate in Sections 3 and 4 that the sweet spot in this trade off is releasing instructions aggressively and speculatively, but by using a path confidence predictor to gate release.

Another part of releasing waitsFor instructions is identifying the correct producer instruction from the dynamic instruction stream in the predecessor thread. Doing this without any help from the compiler is hard. Identifying the program counter of the last-writer is not enough since the same PC may appear multiple times. We make the observation that when the predecessor thread has renamed *all* its instructions, which will be a short while after it reaches its final PC, all last-writers in the predecessor have been renamed. We refer to this specific release point in the dynamic instruction stream as the *potential reconnection point*, or simply *reconnection point*.

This may be a good time to send the physical registers of unsafe registers to the successor thread, all at once and in bulk, so that its waitsFor instructions can be released. We call this scheme *release-on-arrival*(RoA). Note that this release is speculative, since the predecessor may arrive at the reconnection point on a bad path. Another option is to perform release when the predecessor thread has retired all of its instructions, which we call *release-on-retirement*(RoR). We show in section 4.1 that performing RoA while taking branch confidence into account performs much better than RoR, in spite of RoA performing release speculatively.

The final question, of how to validate whether we identified all waitsFor instructions (and did not miss out any), is addressed in Section 3. We augment each thread's register alias table with a set of 4 bits per architectural register to track these inter-thread register dependences.

2.2 Related Work

2.2.1 Dynamic Speculative Multithreaded Systems

Most dynamic speculatively multithreaded processors [1, 10] don't perform explicit register synchronization. Instead, they make the following assumption: Inter-thread register dependences don't occur, and even when they do, the *values* of architectural registers are not changed by any predecessor instruction between the spawn and the reconnection point.

When the above assumption is false, these processors use replay to selectively re-execute waitsFor instructions and their transitive dependents in the successor thread, which usually happens when waitsFor instructions retire. Thus, previous dynamic speculative multithreaded processors have proposed a combination of value prediction and re-execution to solve the out-of-order renaming problem. These systems have produced unique mechanisms that exploit value prediction to get around the renaming problem. However, we show in Section 4.1 that resolving inter-thread register dependences when consumer waitsFor instructions *retire* hurts the performance of a speculative multithreaded processor. Also, our goal is to develop a relatively simple hardware to handle out-of-order renaming. For these reasons, value prediction backed by replay is not an acceptable solution for us.

Skipper [2] is a dynamic out-of-order fetch proces-

sor that fetches from control-independent point in the program when it reaches a hard-to-predict branch. The authors develop an efficient mechanism to rename instructions out-of-order, and we use some of their insights to identify and delay waitsFor instructions. However, Skipper is intended as an add-on to a superscalar processor that *mostly* fetches in-order, whereas our design is intended towards a speculative multithreaded machine where out-of-order fetch is the norm, rather than the exception. For example, our renaming mechanism needs to support out-of-order spawn and reconnect, which Skipper did not investigate. Also note that Skipper performs release when the hard-to-predict branch has resolved, whereas we use a more aggressive confidence based release mechanism. Finally, Skipper's dependence checking mechanism was conservative, in that it sometimes signaled a violation even though a true dependence was not violated, which worked well for their domain. We have found that supporting a non-conservative checking mechanism that flags a misprediction *only* if a violation has occurred to be important.

2.2.2 Compiler-based Speculative Multithreaded Systems

While our goal is to implement register synchronization in a dynamic system without compiler support, we leverage a number of insights from previous work in compiler-based speculative multithreaded processors which place explicit register send and receive instructions in the binary.

The Multiscalar [13] used a compiler to both identify thread boundaries, and to move producer instructions up in the code and consumers (and their dependents) down in the code, as much as possible [15]. The Implicitly Multi-Threaded (IMT) processor [8] extended the Multiscalar to run on top of an aggressive SMT processor with speculation. In this system producer instructions could be fetched and executed along a bad path after a branch mispredict, and thus sometimes release consumers too early, causing some extra flushes. Our work builds on the IMT in three ways. First, we support out-of-order spawn and reconnect of threads. Second, we substantially reduce the probability of flushes caused by producers releasing consumers early by gating the releases with branch confidence information. Third, we have developed a complete system for identifying the consumer instructions dynamically. Since our system identifies release points dynamically, rather than relying on the backward analysis that a compiler could give us, our release points are somewhat conservative compared to those used on the IMT. This last point is discussed further in Section 4.

The Stampede speculative multithreaded system extended the Multiscalar compilation techniques to allow speculative movement of release instructions above statically predicted branches [16], which has interesting parallels to our proposal of performing speculative release based on dynamic branch confidence. Dynamic confidence predictions may give more accurate predictions than the profile based confidence predictor used on Stampede. Further, our system only needs to wait for low confidence branches to execute, rather than retire, which also removes a considerable amount of latency from critical paths. It is difficult to compare our system to a TLS system implemented on a CMP, since our register synchronization is much more tightly coupled.

3 Design

In this section we describe the out-of-order register renaming and synchronization mechanism as implemented in PolyFlow. We start with a broad description of PolyFlow's microarchitecture in Section 3.1. We then provide a detailed view of the renaming mechanism in Section 3.2. We conclude with a discussion of our path confidence predictor in Section 3.3.

3.1 PolyFlow Microarchitecture

Figure 2 depicts PolyFlow's microarchitecture. PolyFlow is a speculative multithreaded processor that dynamically spawns *flows* from a single-threaded application. The overall organization is similar to an SMT machine.

The following sections define a flow, and detail the actions which occur over the lifetime of a flow and their relationship to architectural components.

3.1.1 Flow State

A *flow* is a microarchitectural entity that represents some portion of program execution. Each flow, like a thread, has a current program counter (PC), a rename table (mapping architectural register numbers to physical register numbers), and a reorder-buffer (ROB) of instructions that have been fetched (and possibly completed) but not yet retired. Similar to the POWER5 [12], PolyFlow has a linked-list ROB that is dynamically shared among flows.

However, unlike a thread, a flow's state has a `start pc` (the program counter of the flow's first instruction) and a pointer to the `successor` flow (the next flow in program order). To enable out-of-order renaming, each flow also has a *Diverter Queue*, and four additional bits per RAT entry, which are described in Section 3.2.

The PolyFlow renamer extends the conventional renaming mechanism to work on an out-of-order instruction stream. Each flow uses its own register alias table (RAT) to rename its in-order instruction stream. On dispatch, all flows insert instructions into a shared,

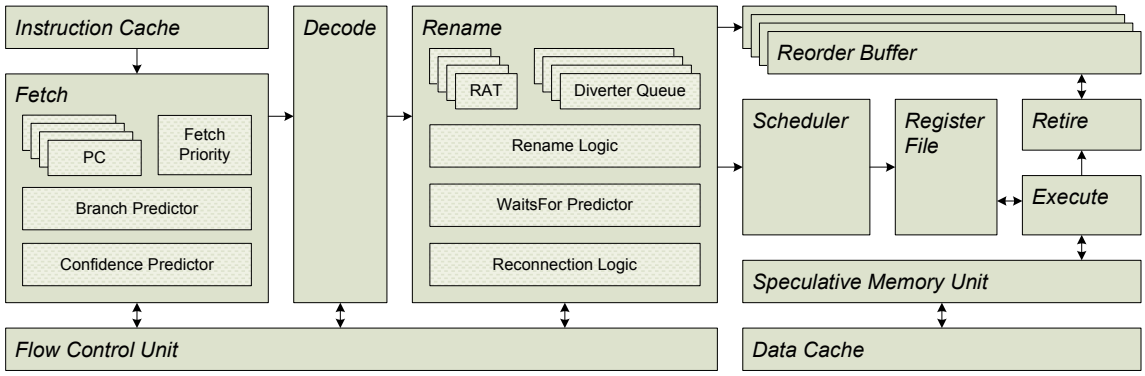

Figure 2: PolyFlow Microarchitecture

non-blocking scheduler, as well as into their own ROB. Thus, intra-flow instruction dispatch in PolyFlow is similar to per-thread dispatch in an SMT processor. As with all other speculatively multithreaded systems PolyFlow's speculative and out-of-order memory system must allow flows to communicate and synchronize memory operands in addition to register operands. Details of our approach to building this memory system are outside the scope (and size constraints) of this paper. Further enhancements to renaming in PolyFlow are discussed in Section 3.2.

3.1.2 Flow Lifetime

The *Flow Control Unit* (FCU) manages the initiation (*spawning*), completion (*reconnection*), and removing (*squashing*) of individual flows.

Flow Spawn. We call the process in which one flow creates a new flow a *spawn*. As instructions are fetched, the FCU identifies control-independent points that could be spawned off. When the FCU decides that a spawn would be profitable (using some heuristics), it *spawns* a new flow on an available SMT context. While some Speculative Multithreaded processors permit only the *youngest* thread (in program order) to spawn new threads, PolyFlow's out-of-order spawn policy allows any flow to spawn new flows. At spawn, the new flow's `start` and `current` PCs are set to the PC of the instruction that the FCU wishes to *spawn to*. An empty reorder-buffer queue is allocated for the successor. The renaming actions that happen at spawn are described in Section 3.2.

Note that flows are chained together in a sequence representing sequential program order, shown in Figure 3. Each flow has a successor flow, which is immediately next to it in program order. When a flow spawns another flow, it inserts the new flow into the sequence between itself and its former successor, as shown in Figure 3 (b).

Reconnection. When a flow's dynamic instruction stream reaches the `start pc` of its successor flow, as in Figure 3 (c), the predecessor flow can *reconnect* with

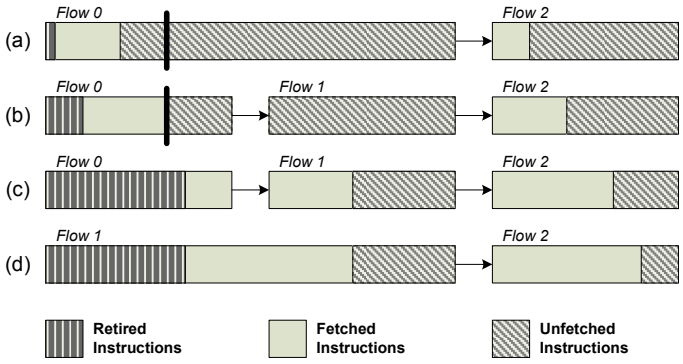

Figure 3: Flow Lifetime: *Flows are represented as the portion of the dynamic instruction stream. Figure (a) shows the state of two flows prior to Flow 0's PC reaching a spawn point, represented by the dark vertical bar. When Flow 0 reaches the spawn point, Figure (b), it spawns Flow 1 which is between Flow 0 and Flow 2 in program order. Figure (c) shows the state of the machine when Flow 0 reaches the spawn PC of Flow 1. At this point, Flow 0 has no more instructions to fetch. At the appropriate time, Flows 0 and 1 are reconnected, shown in Figure (d). After reconnection, all of the instructions once belonging to Flow 0 are now considered part of Flow 1, and the appropriate resources of Flow 0 are freed. Note that only the first flow, in program order, is allowed to retire instructions.*

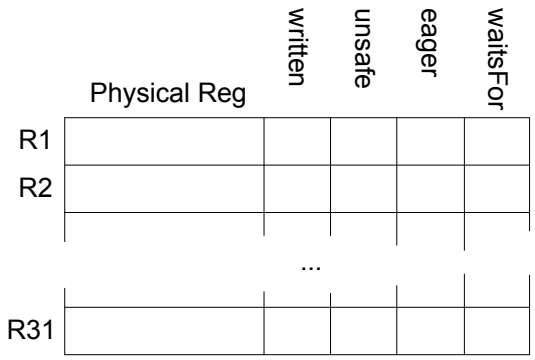

Figure 4: Register Alias Table of a flow.

its successor. At reconnection, the register data flow between the predecessor and successor flows is evaluated (Section 3.2) for correctness. If dependence violations are discovered, reconnection fails and the successor flow is squashed. Otherwise, reconnection succeeds. Successful reconnection effectively *combines* the two flows into one logical flow, associated with a single set of flow state depicted in Figure 3 (d). The resulting flow has one `PC`, `start PC`, and pointer to its successor flow. The reorder buffer of the combined flow is the concatenation of the individual reorder buffers. The tail of the predecessor flow ROB is pointed at the head of the successor flow ROB to build the combined ROB. The rename state of the combined flow is derived in Section 3.2. Note that reconnection can occur only once between any two successive flows and does not need to occur in program order.

3.2 PolyFlow Register Renaming

The goal of our out-of-order renaming design is to support inter-flow register communication in a speculative multithreading system running on an SMT-like pipeline. We insisted that our renaming design support out-of-order spawn and reconnect, because ample previous work has indicated that it was important for performance [4, 1, 9] (we have reconfirmed this in our system). The renaming mechanism supports neither selective reexecution nor value speculation; we deemed it too expensive to add either of these features to support speculative multithreading. Finally, we are careful in our design to make sure that all inter-flow communication is point-to-point because we did not want to build global broadcast buses into the rename unit.

The following sections describe the design of our inter-flow register renaming mechanism, beginning with a description of the additional state we associate with each flow (Section 3.2.1). We then describe how this additional state is updated throughout the lifetime of a flow (Section 3.2.2). Section 3.2.3 then describes our mechanism for identifying registers that are likely to be available to a particular flow at the time of spawn.

3.2.1 Register Renaming Flow State

Two additions to the SMT pipeline to support out-of-order renaming are the *flow rename state bits* and the *Diverter Queues*. To detect and avoid dependence violations we augment the RAT entry of each architectural register in each flow with four bits. These bits are described below and shown in Figure 4.

- **Written:** Indicates that the register has been written by its flow.
- **Unsafe:** Indicates that the flow has a more recent physical register mapping of this architectural register than that copied to its successor (upon the successor spawn).
- **waitsFor:** Registers marked with this bit are expected to be written by a predecessor flow, i.e., the flow should not allow instructions reading this register to execute.
- **Eager:** Set when the register has been *read* by its flow, and was not previously marked `waitsFor` or `Written`.

The per-flow Diverter Queues (Figure 2) are used to hold those instructions for which a correct architectural-to-physical register mapping is currently unknown. The *WaitsFor Predictor* (Section 3.2.3) predicts which architectural registers will be unavailable to a newly spawned flow. In the spawned flow, the renamer delays the execution of instructions that are dependent upon registers marked `waitsFor` by placing them in a per-context Diverter Queue to await renaming. All instructions not explicitly dependent upon waitsFor registers are sent to the scheduler, including the transitive dependents of diverted instructions.

The process of delaying the renaming of instructions that have direct inter-flow dependences is called *diversion*. When the FCU has determined that the diverted instructions in a flow can be safely released, the instructions in the Diverter Queue of that flow are renamed. The released instructions receive correct register mappings from the previous flow for each `waitsFor` register. After an instruction is released, it is sent to the scheduler.

3.2.2 Renaming Flow State Transitions

Flow Spawn. When a flow is spawned, the RAT of the predecessor is copied to the RAT of the newly created successor flow and the four flow state bitmaps are initialized. The successor's `written` and `eager` bits are cleared for each architectural register. The `waitsFor` bitmap is looked up in the WaitsFor Predictor (Section 3.2.3) and logically OR-ed with the predecessor's current `waitsFor` bitmap. The `unsafe` bitmap is inherited from the predecessor, and the predecessor's `unsafe` bitmap is cleared.

```
// Instruction Source Renaming:
instr.src_phys := RAT.phys_reg[instr.src_arch]
RAT.eager[instr.src_arch] :=
  RAT.eager[instr.src_arch] or
  (not RAT.written[instr.src_arch] and
   not RAT.waitsFor[instr.src_arch])

// Instruction Destination Renaming:
instr.dest_phys := allocate_from_freelist()
RAT.phys_reg[instr.dest_arch] :=
  instr.dest_phys
RAT.waitsFor[instr.dest_arch] := false
RAT.written[instr.dest_arch] := true
RAT.unsafe[instr.dest_arch] := true
```

Figure 5: Renaming State Transitions. *The instruction* `instr` *and rename-table* `RAT` *belong to the same flow. Each field in the RAT is updated per instruction source and destination.*

Instruction Rename. The instructions within a flow are seen in-order by the renamer. As they are renamed, the flow state bitmaps associated with the flow are updated to reflect each register read and update. These actions are summarized in Figure 5, described below.

When an instruction renames, each source architectural-to-physical register mappings are found in the flow's RAT and the `eager` bits associated with each source register are updated. The `eager` bit is set if the register is *not* currently `written` or `waitsFor`. Each architectural source's `waitsFor` bit is checked, and if any are set the instruction is steered to the flow's Diverter Queue instead of dispatching to the scheduler.

If the instruction has a destination register, a new physical destination register is assigned from the free register list and the RAT architectural-to-physical mapping is updated as normal. The architectural destination register is marked both `written` and `unsafe`. The final rename action is to clear the destination register's `waitsFor` bit. This indicates that any register which may read from this architectural register in the future should not be diverted, as it will receive the correct mapping.

Violation Detection. When the program counter of a flow has arrived at the `start PC` of its successor, and all of the predecessor instructions have been renamed, PolyFlow may try to reconnect the two flows (Figure 3 (c)). When reconnection is attempted, the bits associated with each architectural register are used to determine if a read-after-write violation has occurred between the two flows.

Since the predecessor's instructions are earlier in program order, we are interested in only the register writes which occurred between point where it spawned the successor flow and its final instruction. This information is held in the predecessor's set of

$$
\begin{aligned}
\texttt{written}_{\text{comb}} &= \texttt{written}_{\text{pred}} \cup \texttt{written}_{\text{succ}} \\
\texttt{unsafe}_{\text{comb}} &= \texttt{unsafe}_{\text{pred}} \cup \texttt{unsafe}_{\text{succ}} \\
\texttt{eager}_{\text{comb}} &= \texttt{eager}_{\text{pred}} \cup \\
&\quad (\texttt{eager}_{\text{succ}} - \texttt{written}_{\text{pred}}) \\
\texttt{waitsFor}_{\text{comb}} &= \texttt{waitsFor}_{\text{pred}} - \texttt{written}_{\text{succ}}
\end{aligned}
$$

Figure 6: Rules for combining the information sets of two flows, where the *pred* flow precedes, and is reconnecting to, the *succ* flow to form a combined flow *comb*.

`unsafe` bits. A violation can only occur if the successor flow read from a register *before* it wrote to it, which is captured by the successor's `eager` bits. The intersection of these two bit sets represent the architectural to physical mappings that the successor read from incorrectly.

To check whether the two flows can be correctly reconnected we check the condition:

$$\texttt{unsafe}_{\text{pred}} \cap \texttt{eager}_{\text{succ}} == \emptyset$$

If the intersection of the predecessor's `unsafe` set and the successor's `eager` set is empty, then we have guaranteed that the successor accessed register mappings for architectural registers that were either (1) not modified between the spawn point and the rename point or (2) modified by the successor before the read (so the source register was renamed correctly). Note that correctly predicting the `waitsFor` set is the key to avoiding reconnection check failure, since those registers marked `waitsFor` will not be read from eagerly. A failed reconnection results in the squashing of the successor flow, and all flows which follow. The predecessor flow resumes fetching the instructions which had belonged to the successor flow, as if the successor had never been spawned.

Flow Reconnection. If reconnection is successful, we want to combine the two flows into one. The key point is that the resulting flow should appear as if there had *never been two flows*. For simplicity, we eliminate the predecessor flow and transform the successor flow into the combined flow. The combined flow will have the current `PC` of the successor flow, since this `PC` represents the only instructions in the two flows which remain to be fetched. The `start PC` flow is the `start PC` of the predecessor flow, since this is the first `PC` fetched either flow in program order.

The information from each flow's RAT also needs to be merged. We make use of our register state bits to construct the resulting RAT. Since we will use the RAT of the successor flow as the base for the combined flow's RAT, we need only copy from the predecessor those mappings which were modified in predecessor and *not* the successor. All other register map-

pings are either correct because they are unmodified in both flows (and thus identical) or have only been modified in the successor. The result is an architecturally correct RAT, associated with the combined flow's current state. The set of registers updated in the successor's RAT upon reconnection is:

$$\texttt{unsafe}_{\text{pred}} - \texttt{written}_{\text{succ}} = \texttt{updated}_{\text{comb}}$$

In the case of the four bitsets, we want the result to appear as if the two flows have never been separate. The details of combining of the bitsets are given in Figure 6. The `eager` and `waitsFor` bits are discussed below for added clarity.

The `eager` set of the combined flow should represent the set of architectural registers that would have been read by the combined flow before they were written by the combined flow. Thus, the `eager` set of the combined flow will be the union of the `eager` set of the predecessor flow and those `eager` registers of the successor that were not also written by the predecessor flow.

The `waitsFor` set of the combined flow should represent the set of architectural registers that the combined flow should *still* is unlikely to have the correct mapping. Since every register mapping that the predecessor flow is aware of has been communicated to the successor flow (the RATs have been merged), the set cannot be larger than the set of registers that the predecessor flow was waiting for. However, some of the predecessor's `waitsFor` registers may have been redefined by the successor flow, in which case, they needn't be 'waited for'. Thus, the `waitsFor` set of the combined flow is the `waitsFor set` of the predecessor flow less the `written` set of the successor flow.

The final step in reconnection is to process the Diverter Queues of the two flows. Each instruction in the predecessor's Diverter Queue will remain diverted. However, the instructions in the successor's Diverter Queue can potentially be renamed since the predecessor may have generated the register mapping on which they are dependent. For each instruction in the successor Diverter Queue, we look up the source architectural registers we were waiting for in the RAT. If those entries are still marked `waitsFor` in the RAT then the instruction is *remains* diverted in the combined flow. Otherwise, the instruction now has the source architectural-to-physical register mapping it was waiting for and is thus renamed and dispatched to the scheduler.

3.2.3 WaitsFor Prediction

A dynamic out-of-order renaming system has to predict the set of registers that the predecessor flow will write between the spawn point and the reconnection to the successor flow. We have coined these registers the spawn's *waitsFor* set. When a flow is spawned, it receives a *predicted* `waitsFor` set from a hardware structure called the *WaitsFor Predictor*.

A "spawn" in a Speculative Multithreaded processor can be uniquely identified by a pair of program counters: the PC of the instruction which triggered the spawn and the PC of the first instruction of the spawned flow. The WaitsFor Predictor has a table with one entry per spawn PC pair. The prediction returned for a spawn is a bitmask which represents, for each architectural register, whether or not the register is expected to be written by the predecessor as it executes instructions between the spawner PC and spawned PC.

The key insight behind the WaitsFor Predictor is that, while the actual control flow path taken by the predecessor flow in going from the spawn point to the reconnect point may change dynamically, the set of registers that are written does not vary significantly. The waitsFor Predictor, needs to be highly accurate: false positives cause instructions in the successor thread to wait unnecessarily, while false negatives (may) cause dependence violations.

To decide which registers should be marked `waitsFor`, the predictor keeps a counter per architectural register for each spawn PC pair. As the predecessor thread fetches instructions, the `unsafe` bitmap (described earlier) keeps track of registers that *should have been* marked `waitsFor` in its successor. When the predecessor arrives at the reconnection point, the predictor is trained using the `unsafe` bitset as the true `waitsFor` set.

If a particular register was `waitsFor`, its corresponding counter is incremented by a static *upcount* value. Otherwise, register's the counter is decremented by *downcount*. The next prediction compares counter values against a *threshold*, to decide if a register should be marked `waitsFor`. We present results of three different WaitsFor Predictor configurations in Section 4.2. It was observed that a predictor using 1-bit counters per register (with an upcount, downcount, and threshold of 1) gives the best performance overall.

3.2.4 Example

Figure 7 illustrates the use of rename bitsets as threads get created, reconnect or get squashed, using machine snapshots at five different times, A to E. In the beginning F0 is the only active thread, with architectural register R1 mapped to physical register 101, and R2 mapped to 103. Upon fetching and decoding past instruction S2, it spawns the flow F2. When the instruction S2 is renamed, F2 inherits its RAT from F0. In particular, the mappings for R1 and R2 are copied over to from F0 to F2's RAT. In addition, as Section 3.2.2 describes, F0's unsafe vector is copied over to F2 and cleared (although the unsafe vector of F2 is not functionally useful-it does not have a successor flow). Ea-

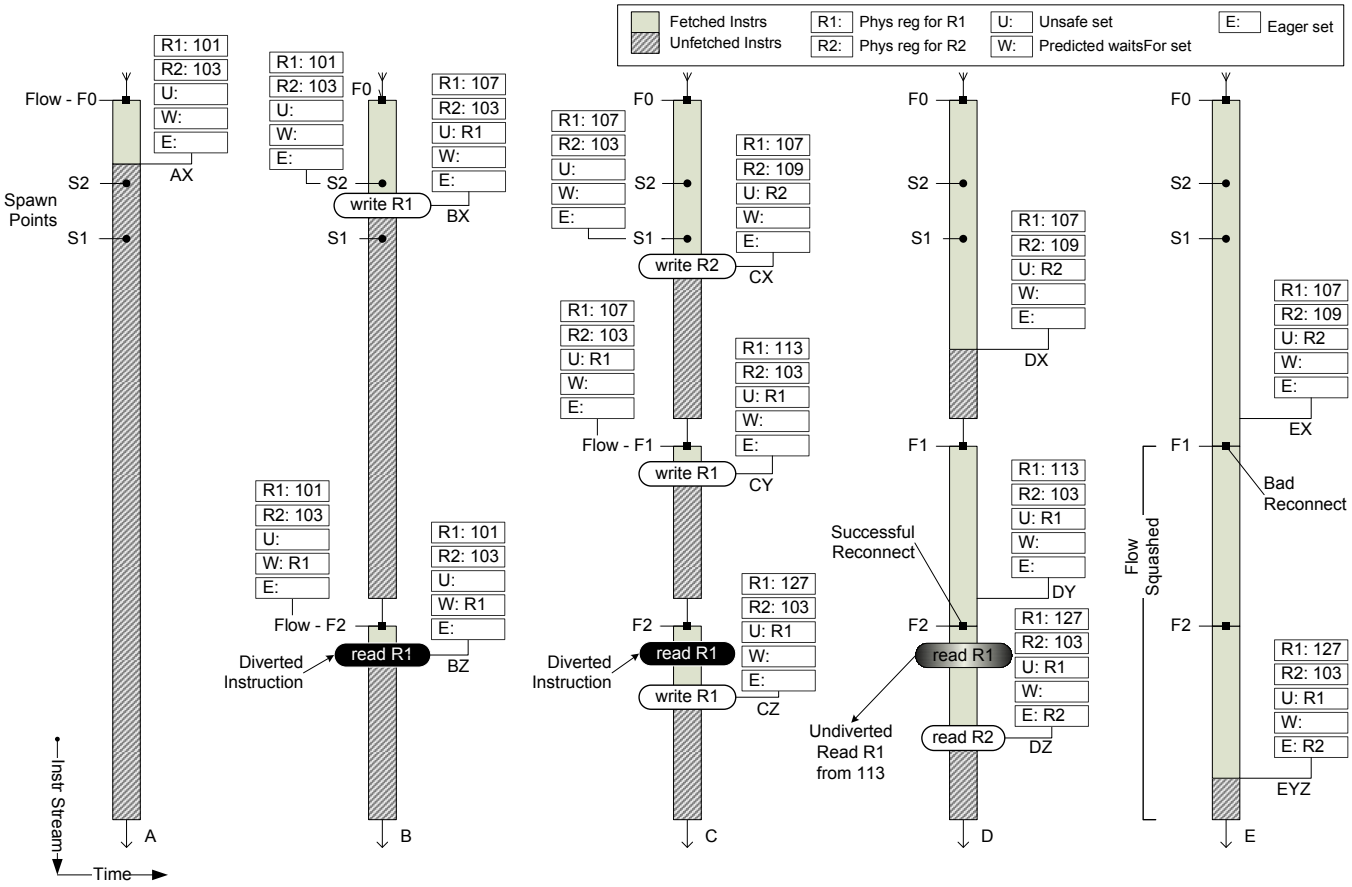

Figure 7: An example illustrating the use of rename bitsets. Initially there is a single flow F0, which spawns flow F2 and then F1. Flow F1 reconnects successfully to F2. Subsequently F0 reconnects to the combined flow, signaling a misreconnect. The progress in time is show along X-axis

ger set of F2 is initialized as being empty. Register R1 is predicted as waitsFor, and any instruction having R1 as one of its sources is diverted pending dataflow from a predecessor.

At time period B, flow F0 writes to register R1, marking it as unsafe - if successor flow F2 were to use the mapping of R1 copied over from F0 at spawn, it would have used the incorrect mapping. However, since F2 had R1 marked as waitsFor, it correctly diverted any instructions which read from R1 (note that the instruction reading R1, marked in black, has infact been diverted). Next F0 spawns F1, which is predicted to have an empty waitsFor set. Initialization of bit vectors proceeds as for previous spawn. In particular, the unsafe set is copied over from F0 to F1, F0's unsafe set is cleared, and F1's unsafe set gets R1. Note that register R1 is unsafe for flow F1 since this flow has a more recent mapping for R1 than the flow's successor, F2. At time C, F0 writes to R2, and F1 writes to R1, causing them to be marked unsafe in their respective flows. Also, F2 writes to R1, causing R1 to be *removed* from its waitsFor set, since any future instructions will get the correct mapping for R1, and added to its unsafe set.

Upon reaching the reconnection point, flow F1 tries to reconnect to F2. The reconnect checks are done, and since F2 didn't eagerly read any register that was unsafe in F1, the reconnect is successful. The new flow's unsafe vector is the union of the two unsafe vectors, which is R1. The new flow gets an empty waitsFor set, since the waitsFor set of flow F2 was empty. Also note that the instruction reading R1 that was diverted in F2 now undergoes undiversion and gets the correct physical register mapping, 109. The combined flow makes progress, and at time D, does an eager read of register R2, based on the mapping inherited from F0. This mapping is wrong, since R2 is marked unsafe in F0. Thus, when F0 reaches the reconnection point at time E, reconnection checks fail for register R2. An invalid merge is signaled, and the successor thread formed from the merge of F1 and F2 is flushed.

3.3 Path Confidence Prediction

Since it is possible for the predecessor flow to reach reconnection along a misspeculated path, not all "successful" reconnections result in a leap in forward progress. Up until reconnection, branch misspeculations in the predecessor do not effect successor flows;

we can safely roll back the state of the flow to the misspeculation point without affecting any other flows.

However, if a flow contains a misspeculated branch and *has* reconnected with another flow, then the combined flow is the only flow we have to work with. We must roll back to the state at the time of the mispredicted branch, even if the instructions previously associated with the successor flow had no dependences on those instructions along the mispredicted path. This results in a loss of a significant amount of computation that could be avoided by *delaying reconnection until the machine has a high probability of being on the predecessor's correct path.* To this end, we use a branch confidence predictor to estimate the likelihood that a flow contains unresolved and mispredicted branches.

Branch confidence predictors [5, 3] estimate the probability that a branch is predicted correctly. When a flow reaches the possible reconnection point, we would like to estimate the likelihood that all of its *unresolved* branches were predicted correctly. In other words, we need the cumulative confidence estimate for all unresolved branches in the flow. We call this cumulative estimate *Path Unconfidence*. A high value of path unconfidence indicates uncertainty about the flow's unresolved branches. When a flow fetches a branch and predicts its direction, a confidence predictor provides an estimate of how likely is the branch to be mispredicted, which we call the *branch unconfidence*. This value is added to the path unconfidence of the flow that fetched the branch. When branches resolve, the corresponding flow's path unconfidence is decremented by the branch's unconfidence.

Path unconfidence is used to *gate* the reconnection process: when a flow arrives at the reconnection point, we allow it to reconnect to its successor only if its path unconfidence is below a certain threshold, called the *Reconnection Threshold*. Otherwise, the flow waits at the reconnection point, until one of the following three things happen:

First, if the flow executes a mispredicted branch, the flow recovers from the misspeculation as normal, continuing along its new path without affecting its successor flow. Secondly, if the flow executes a branch that was correctly predicted which lowers the flow's path unconfidence below the reconnection threshold, the reconnection is allowed to proceed. Lastly, if a *timeout* number of cycles pass while waiting to reconnect, reconnection is triggered in spite of the current path unconfidence. Using a timeout value helps to improve performance, since flows often have unresolved branches which are sitting in the Diverter Queue.

To build confidence mechanisms for reconnect-gating, we leveraged previous work in branch confidence estimation, along with extensive experimentation to determine the kind of predictors that well in this domain. This has resulted in a unique mechanism for determining path unconfidence which we describe next. However, providing detailed reasons for our choices is beyond the scope of this paper.

Parameter	**Value**
Pipeline Width	8 instr/cycle
Branch Predictor	8Kbit gshare
Confidence Predictor	8Kbit JRS
Misprediction Penalty	at least 8 cycles
Reorder Buffer	1024 entries, dynamically shared
Functional Units	8 identical fully pipelined units
L1 I-Cache	8Kbytes, 2-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	8Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 100 cycle miss

Figure 8: Pipeline parameters.

We use two different branch confidence estimators, working together: the enhanced JRS predictor [5, 3] with 4 bits per entry, and another estimator, which we call the Global Miss Distance Counter (GMDC). The GMDC contains one 4-bit counter per flow to keep track of the number of branches that have been fetched since the most recent mispredict was resolved. This estimator exploits the insight presented in [3] that branch mispredicts are often clustered together, and thus, branches fetched immediately after a mispredict should have a lower confidence.

We keep track of the path confidence from these two estimators separately, in two different registers, called *JRS Path Unconfidence*, and *GMDC Path Unconfidence*. The JRS unconfidence value assigned to a branch is the counter value read from the JRS predictor subtracted from 16. (Note that this implies that even the most confident branches get an unconfidence value of 1.) This unconfidence value is added to the flow's JRS path unconfidence. Similarly, the GMDC unconfidence value for a branch is the value of the flow's GMDC counter, subtracted from 16. To gate reconnect, we use two separate reconnection thresholds, a *JRS reconnection threshold*, and an *GMDC reconnection threshold*. Both the JRS path unconfidence and the GMDC path unconfidence of a flow must be below their respective thresholds before the reconnection is allowed.

4 Evaluation

Our experimental evaluation was performed on a simulator for the PolyFlow Speculative Multithreaded architecture. The simulator executes a variant of the 64-bit MIPS instruction set ISA which does *not* have

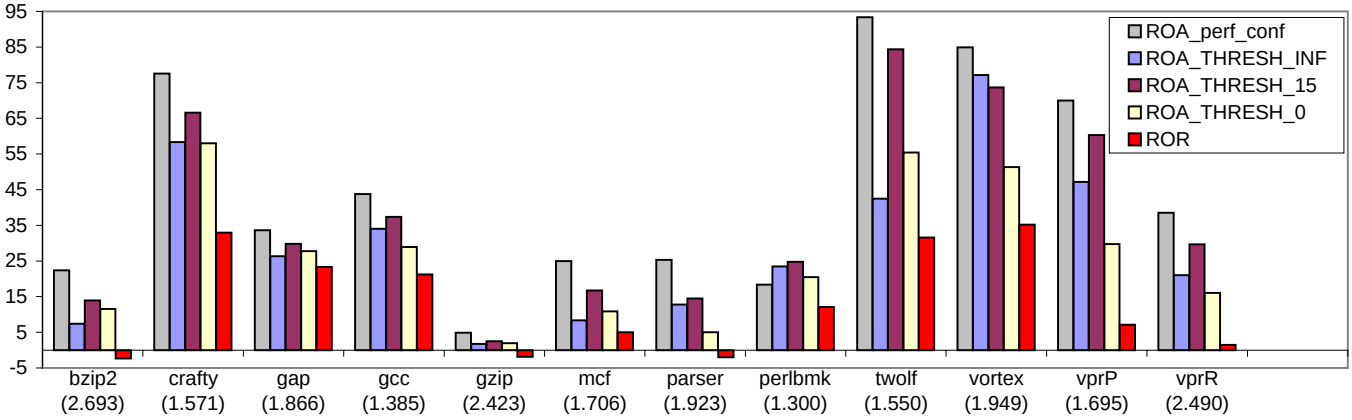

Figure 9: The impact of path confidence information. *The y-axis shows percentage speedup of speculative multithreading over a superscalar (base superscalar IPC shown in parenthesis). The leftmost bar shows speedup of release at arrival given an ideal (oracular) confidence predictor. The second bar shows a "hyper-aggressive" policy where release is performed whenever we arrive at the potential reconnection point, disregarding confidence information. The middle bar shows the use of a realistic confidence predictor with a threshold set at 15. The fourth bar shows performance degradation due to a non-aggressive policy that releases only when there are no remaining unexecuted branches. The final bar shows the slow down from an even less aggressive policy that releases only when all branches have retired.*

any special instructions to support speculative multithreading. The PolyFlow simulator is fully execution driven. It not only simulates timing, but also *executes* instructions out-of-order in the backend, writing results to the register file out of program order. When an instruction is retired, its results are compared against an architectural simulator, and an error is signaled if the results don't match. The PolyFlow simulator models mispredicted instructions accurately, since the backend treats good path and bad path instructions in exactly the same way: both types of instructions execute and write values to the physical register file, and we use rename checkpoints to recover from branch mispredictions. The simulator also renames instructions out-of-order speculatively and uses the bitmap based checking mechanism described earlier to track waitsFor instructions and catch true dependence violations in the presence of out-of-order spawns and reconnections.

Many essential functions of the PolyFlow architecture are topics of ongoing research, including the spawn policy, the memory system, and out-of-order branch and confidence prediction. For the purposes of this paper, we idealized these parts of the machine, so that we could focus on the performance effects of renaming and data-forwarding. Thus, the simulator uses oracular memory dependence prediction. The spawn points we use are obtained from a control-independence analysis of program traces, and the spawn policy uses oracularly known distance metrics to decide which spawns are useful. Finally, we used branch direction and confidence predictions from GShare and JRS [5] predictors respectively, executing the program in-order (i.e., we did not model out-of-order branch resolution while training these predictors).

We simulate a very aggressive, 8-wide machine, running 8 threads, with the configuration given in Figure 8. The superscalar model that we use is capable of fetching a maximum of one taken branch per cycle. In PolyFlow mode, the machine can fetch from two threads in a cycle, with a maximum of one taken branch per cycle per thread. The PolyFlow instruction fetch unit uses path confidence to prioritize among different threads, giving preference to threads that have a higher confidence value. However, note that the results in Figure 10, use a round-robin fetch policy.

In Section 4.1 we demonstrate the performance impact of forwarding data between flows speculatively, but only when we have high confidence in the speculation. In particular we find that our path-confidence predictor can achieve speedups over a base superscalar that are within 10% of an oracular system that synchronizes at the "perfect" time. Section 4.2 demonstrates that our waitsFor predictor performance is also nearly ideal.

In the results presented here, we fast forwarded through the initialization phase of all benchmarks, and executed 100 million instructions after that. All the graphs that we present show the speedup of different Speculative Multithreaded configurations over a superscalar. The absolute IPC numbers for the superscalar are shown below each benchmark name in Figure 9.

4.1 Speculative Data Forwarding

In this section, we look at the effects of speculative data-forwarding on the performance of a PolyFlow system. We use a perfect waitsFor predictor in these experiments. The results with a real waitsFor predictor are presented in Section 4.2. In Figure 9 we compare a variety of policies for selecting the time at which producer threads release the consumer threads.

Most of these are Release-on-Arrival (RoA) policies, where inter-thread data forwarding (by releasing the waitsFor instructions) happens when the predecessor arrives at the potential reconnection point. The policies differ in how aggressive they are about assuming that we have arrived at the reconnection point along a good branch path instead of a mispredicted branch path. We model four different RoA policies:

- RoA, Perfect Branch Confidence: In this policy, data is forwarded from the predecessor to the successor on good path arrival at the reconnection point. Good path arrival is determined oracularly. This configuration is the upper bound on the performance of RoA.
- RoA, Path Confidence Threshold 15+10: This policy uses the path confidence predictor described in Section 3.3 to predict whether arrival at the reconnection point is good path or bad path. We used a JRS unconfidence threshold of 15, and a GMDC unconfidence threshold of 10.

  A predecessor thread releases consumers in its successor thread when the predecessor thread arrives at the potential reconnection point, and the path unconfidence (both JRS and GMDC) are less than the threshold. If the path unconfidence is too high the predecessor waits till either its branches resolve, decreasing its path unconfidence, or 35 clock cycles elapse, which ever is earlier. At this point, waiting consumer instructions in the successor are released.
- RoA, Path Confidence Threshold Infinity: This policy does not use path confidence, and aggressively forwards data from predecessor to successor immediately upon the predecessor's arrival at the potential reconnection point.
- RoA, Path Confidence Threshold Zero: This configuration conservatively forwards data from predecessor to successor only when all branches in the predecessor have resolved (completed execution). Thus, reconnection happens non-speculatively.

The final configuration we evaluate is Release-on-Retirement (RoR), which is even more conservative than RoA with Path Confidence Threshold Zero. This policy waits to forward from predecessor to successor until the predecessor has retired all its instructions, and therefore, is forwarding completely non-speculatively. Note, however, that RoR is somewhat less conservative than would be a policy based on waiting until the *consumer* instruction retired, as would happen in systems that base their synchronization on full value speculation with validation, and partial reexecution, at retirement [1, 10].

Figure 9, demonstrates that forwarding data aggressively and speculatively gives better performance than forwarding it conservatively. Release-on-Retirement is particularly bad, and results in a small slowdown over the superscalar for some benchmarks. The other configurations, RoA with a threshold of infinity, and RoA with a threshold of 0, both perform significantly worse than RoA with perfect confidence, although there is no clear favorite among the two. For some benchmarks, like *twolf* and *bzip2*, waiting until all branches in the predecessor have resolved is better. For other benchmarks, like *vortex* forwarding data immediately upon arrival is better.

RoA with a threshold of 15+10 performs better than the above two configurations, and comes close to RoA with perfect confidence. We have also found that the particular path confidence threshold that performs best varies from one benchmark to another, although the results presented here are with a fixed threshold. An adaptive algorithm that adjusts the threshold dynamically would probably do even better.

Note that using confidence to gate the reconnect signal reduces performance for one benchmark (vortex), compared to forwarding data immediately upon arrival. The reason for this is the confidence based fetch prioritization policy used in our simulations. Such a policy reduces the likelihood of bad path arrival at the reconnection point: threads that are low in confidence fetch fewer instructions, and thus, are less likely to arrive at the reconnection point. With a different fetch prioritization policy, using branch confidence to gate data forwarding becomes *more* important. For example, Figure 10 demonstrates that in a system with a *round-robin* fetch algorithm, a threshold of 15 always performs significantly better than thresholds of zero or infinity. In this case all benchmarks, including vortex, are helped by using branch confidence to gate synchronization.

Recall that our Release-on-Arrival mechanism forwards data from a predecessor thread to a successor thread only after the predecessor has arrived at the potential reconnection point. We wanted to understand the performance cost of this design decision since several compiler based speculative multithreading systems [8, 16] have taken pains to release individual registers at the earliest point where the compiler can prove there will be no more modifications to that register. Thus, we also performed experiments where we compared our Release-on-Arrival policy with a completely

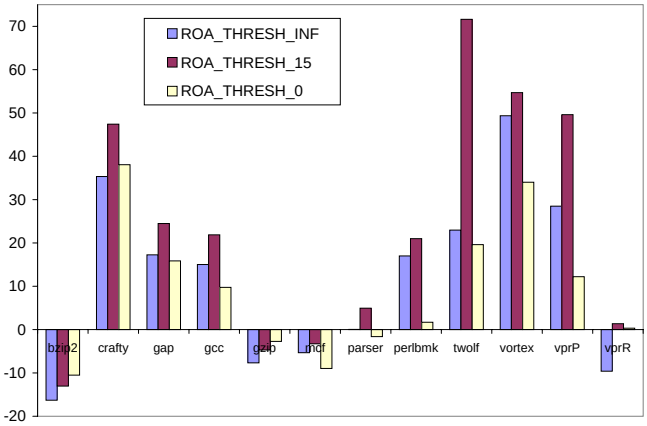

Figure 10: Using path confidence information to gate aggressive synchronization is even more important when the fetch policy is not biased toward more confident paths. *This graph shows speculative multithreading speedups over superscalar with a (sub-optimal) round-robin fetch policy. In this case the "hyper-aggressive" synchronization policy* never *beats the confidence gated policy.*

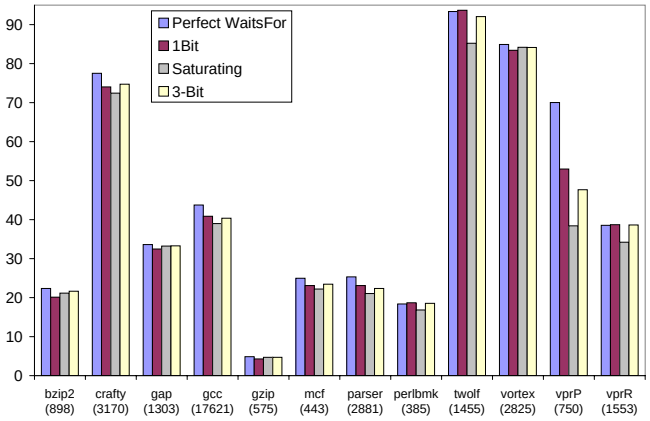

Figure 11: A one bit up-down waitsFor predictor usually provides speedups over a superscalar that are within a few percent of those produced by an oracle waitsFor predictor. *The y-axis shows percentage speedup of release-on-arrival with a variety of waitsFor predictors over the aggressive superscalar. The total number of spawner-spawnee pairs (predictor entries) is shown for each benchmark along the x-axis. The leftmost bar shows an "oracle" waitsFor predictor. The second bar shows a 1-bit predictor that simply uses the* `unsafe` *set from the previous instance of this spawn. The third bar shows a conservative saturating "up-only" counter that gets set if a particular register should* ever *have been made waitsFor in the past. The rightmost bar shows a 3-bit counter.*

unrealizable oracle that can forward data from a producer instruction to all consumer instructions as soon as the dynamic producer *instruction* is renamed. Note that this may be considerably earlier than a compiler could place a release or send instruction, since we are working with the dynamic instruction stream rather than the static program. We found, nonetheless, that Release-on-Arrival was usually quite competitive with the unrealizable oracle. For 8 out of our 12 benchmarks (bzip2, crafty, gap, gzip, parser, perlbmk, vortex and vpr-route) the unrealizable oracle got less than 10% additional speedup over that achieved by Release-on-Arrival. For the other four benchmarks (gcc, mcf, twolf, and vpr-place) there is room for improvement. The twolf benchmark, in particular, achieved an extra 38% speedup (132% compared to RoA's 94%) over the superscalar when synchronization is performed by the unrealizable oracle.

4.2 WaitsFor Prediction

In this section, we evaluate design space of waitsFor prediction. For all the experiments in this section, we use Release-on-Arrival with perfect branch confidence as the data-forwarding strategy, so that we can focus on the performance of the waitsFor predictor.

Recall that the waitsFor predictor decides which instructions in the successor thread should be delayed. If the predictor fails to mark an instruction as waitsFor, a dependence violation and thread squash could happen. If the predictor marks instructions as waitsFor unnecessarily, instructions in the successor thread may be delayed waiting for synchronization that is not actually required.

We implemented a number of different waitsFor predictors, with different values for upcount(U), downcount(D) and threshold(T). We examine the performance of three different predictors: a 1-bit predictor that remembers the true waitsFor set from last time(U=1, D=1, T=1); a saturating predictor that never counts down (U=1, D=0, T=1); and an 3-bit predictor (U=8, D=1, T=1). Their performance is shown in Figure 11, which also shows a perfect predictor for reference. We find that except for one benchmark (vpr-place), the amount of hysteresis in the waitsFor predictor does not affect performance much.

For vpr-place, a 1-bit predictor that simply remembers the waitsFor set from the last time works best, but still loses about 13% of the speedup achieved by the oracle predictor. The fact that the 1-bit predictor works better than the "up-only" and 3-bit predictors indicates that this application has a consumer on the critical path that only occasionally needs to be synchronized. The non-oracular waitsFor predictors conservatively make this consumer waitsFor too often.

The total number of entries in the predictor, i.e., the total number of unique spawnerPC-spawnedPC pairs, is shown below each benchmark in Figure 11. We did not model size constraints and replacement policies for the waitsFor predictor.

5 Conclusion

This paper describes the inter-flow register renaming and synchronization hardware of an aggressive speculatively multithreaded system. The system runs on top of a simultaneous-multithreading-like pipeline that can support up to 8 simultaneously active threads. Our system combines a novel set of features. First, our inter-flow renaming and synchronization scheme supports out-of-order spawn and reconnect of threads. Second, it aggressively and speculatively synchronizes to minimize latency added to the critical path. Third, we have designed a path confidence predictor that works particularly well to gate our synchronization scheme so that it is not too aggressive. Fourth, we have demonstrated that our path confidence gating mechanism gives us performance within 10% of an oracle that "magically" knows the perfect time to perform synchronization. Finally, we have demonstrated that a straight-forward prediction of a single bit per architectural register allows us to near-optimally identify the set of consumer instructions with no compiler support at all.

We made several decisions early in the design process. In particular, we decided to target our design at a tightly coupled (simultaneously multithreaded) style system rather than a CMP. Also, we decided that our system would run binaries "out of the box," dynamically discovering inter-thread synchronization points, rather than relying on a compiler to identify and reschedule synchronization instructions. We believe that the insights we have gained in our system may well apply in these two broader contexts.

In particular, we believe that our insights about the necessity of aggressive, speculative, and path confidence gated synchronization will carry over to speculative multithreading systems built on top of CMPs. Our results also indicate that there may be some benefit to be gained by identifying the last dynamic register producer along particular paths of the program, and we plan to investigate mechanisms, both dynamic and compiler based, to gather this information.

Acknowledgements

The work reported in this paper was supported in part by the National Science Foundation under grant CCR-0429711. Computational resources were supported by an equipment donation from AMD Corp., and the National Science Foundation under grant EIA-0224453. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship (Sam Stone). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *31st Int'l Symp. Microarchitecture*, pages 226–236, November 1998.

[2] Chen-Yong Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO 34*, pages 4–15, 2001.

[3] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew R. Pleszkun. Confidence estimation for speculation control. In *ISCA*, pages 122–131, 1998.

[4] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS VIII*, pages 58–69, October 1998.

[5] Erik Jacobsen, Eric Rotenberg, and James E. Smith. Assigning confidence to conditional branch predictions. In *MICRO 29*, pages 142–152, 1996.

[6] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[7] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Int'l Conf. Supercomputing*, pages 77–84, 1998.

[8] Il Park, Babak Falsafi, and T. N. Vijaykumar. Implicitly-multithreaded processors. In *ISCA-30*, pages 39–51, 2003.

[9] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *19th Int'l Conf. Supercomputing (ICS)*, pages 179–188, 2005.

[10] Eric Rotenberg and James E. Smith. Control independence in trace processors. In *International Symposium on Microarchitecture*, pages 4–15, 1999.

[11] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *HPCA 7*, January 2001.

[12] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[13] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA 22*, pages 414–425, June 1995.

[14] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA 4*, pages 2–13, February 1998.

[15] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, January 1998.

[16] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X*, October 2002.