Appears in the 34th Annual International Symposium on Microarchitecture (MICRO), pages 4-15, December 2001.

Skipper: A Microarchitecture For Exploiting Control-flow Independence

Chen-Yong Cher and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University, {chenyong,vijay}@ecn.purdue.edu

Abstract

Although modern superscalar processors achieve high branch prediction accuracy, certain branches either are inherently difficult to predict or incur destructive interference in prediction tables, causing significant performance loss due to mispredictions. We propose a novel microarchitecture, called Skipper, to handle such difficult branches by exploiting control-flow independence. Previous approaches to handling difficult branches, one way or another, amount to executing incorrect instructions, squandering cycles and resources such as the i-cache bandwidth. Skipper altogether avoids incorrect instructions by skipping over, without even fetching, the control-flow dependent computation conditioned by a difficult branch. Instead, Skipper fetches and executes the control-flow independent instructions, which are past the point where the branch's taken and not-taken paths reconverge, and which need to be executed irrespective of the branch outcome. Because Skipper executes the correct control-flow dependent instructions after the difficult branch is resolved, it conserves the valuable resources.

Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in a superscalar pipeline. Skipper fetches the skipped control-flow dependent instructions after the post-reconvergent instructions, out of program order. We describe key mechanisms to implement Skipper without unduly complicating the pipeline despite out-of-order fetch. SPECint95 simulations show that Skipper performs 10% and 8% better than superscalar and the previously-proposed Polypath, respectively, when all three microarchitectures have equal i-cache bandwidth and hardware resources.

1. Introduction

Modern processors employ branch prediction to avoid pipeline stalls caused by branches. The microarchitecture community has made impressive improvements in prediction accuracy [15,25]. Nevertheless, achieving perfect prediction may be difficult because certain branches either are inherently hard to predict or incur destructive interference in finite-sized prediction tables. Mispredictions of such "difficult" branches cause considerable performance loss, and will continue to do so in the future.

We propose a novel microarchitecture, called Skipper, to handle difficult branches. Skipper avoids predicting difficult branches by skipping over the computation conditioned by such a branch, and exploits the fundamental property of *control-flow independence* [17]. The computations in a branch's taken and not-taken paths are conditioned by the branch and are *control-flow dependent* on the branch because whether each of the computations gets executed or not depends on whether the branch is taken. In contrast, the computation immediately following the point where the branch's taken and not-taken paths reconverge is *control-flow independent* of the branch because the *post-reconvergent* computation gets executed irrespective whether the branch is taken or not. A previous study shows potential speedups of about 30% in a wide-issue superscalar by exploiting control-flow independence [17].

Previous approaches to handling difficult branches are: (1) to execute both the taken and not-taken paths conditioned by a difficult branch [12,24,11] or (2) upon a misprediction, selectively recover control-flow independent instructions by not squashing the data independent instructions, and re-executing only the data dependent instructions [17]. Because the first approach executes both paths one of which is incorrect and the second approach executes incorrect instructions from the mispredicted path, both approaches squander cycles and valuable resources such as the i-cache bandwidth. Incorrect instructions are numerous because they include not only incorrect control-flow dependent instructions but also control-flow independent instructions which are data dependent on the incorrect control-flow dependent instructions. In [17], the proponents of the second approach conclude that "the biggest performance limiter is wasted resources consumed by incorrect control dependent instructions".

To conserve the valuable resources, Skipper altogether avoids incorrect instructions by skipping over, *without even fetching,* the control-flow dependent instructions (both taken and not-taken paths) conditioned by a difficult branch. Skipper fetches and executes the post-reconvergent instructions, which need to be executed irrespective of the branch outcome, and executes *only* the correct control-flow dependent instructions *after* the difficult branch is resolved. Skipper is the first proposal to exploit control-flow independence by skipping over control-flow dependent computation in the context of a superscalar pipeline. Superscalars employ sophisticated out-of-order instruction-issue techniques which routinely skip over data dependent instructions but *not* control-flow dependent instructions. Other approaches, employing hardware and software assists vastly different than a superscalar, skip over instructions to pursue multiple flows of control: Multiscalar [20] uses the compiler to identify reconvergence points, Dynamic Multithreading [1] uses hardware to skip over loops and calls, but not branches.

Unlike superscalars which always fetch instructions in (predicted) program order, Skipper fetches the skipped control-flow dependent instructions *after* the post-reconvergent instructions, out of program order. Thereby, Skipper exploits control-flow independence of the post-reconvergent computation, and overlaps execution of computation before the branch (and resolution of the difficult branch) with the execution of the post-reconvergent computation. Execution overlap comes from post-reconvergent instructions that are data independent of the skipped instructions. Skipper forces the post-reconvergent instructions, that are data dependent on the yet-to-be-fetched skipped computation, to wait till the difficult branch is resolved and the correct path within the skipped computation is fetched and executed. Note that conventional superscalars delay *all* instructions following a mispredicted branch till the instructions are re-executed. In contrast, Skipper delays *only* the skipped instructions and the post-reconvergent data dependent instructions, but does *not* delay the post-reconvergent data-independent instructions.

We describe four mechanisms to implement Skipper in an out-of-order pipeline: First, to identify difficult branches, Skipper uses the previously-proposed JRS scheme [9]. Second, to determine difficult branches' reconvergent points, Skipper employs a heuristic based on idiomatic control-flow code patterns generated by modern compilers for conditional constructs, without requiring scanning of instructions as in [17]. Third, despite out-of-order fetching of the skipped instructions, Skipper maintains program order in the instruction window and the load/store queue. On fetching a difficult branch, Skipper creates an appropriately-sized, contiguous gap in the instruction window and the load/store queue, to be filled later by the skipped instructions from the correct path. Fourth, to force data-dependent post-reconvergent instructions to wait till the yet-to-be fetched skipped instructions execute, Skipper estimates register dependencies, learning from prior dynamic instances. At the time of skipping, Skipper updates the register rename tables using this dependence information, making post-reconvergent data-dependent instructions wait.

The main contributions of this paper are:

- we propose Skipper, the first proposal to skip control-flow dependent instructions, without wasting resources on incorrect control-flow dependent instructions.
- we describe key mechanisms to implement Skipper without unduly complicating the pipeline despite out-of-order fetch.
- SPECint95 simulations show that Skipper performs 10% and 8% better than superscalar and Polypath, respectively, when the three architectures have equal i-cache bandwidth and hardware resources.

In Section 2, we discuss how Skipper is mapped to a superscalar microarchitecture at a high level. In Section 3, we describe the pipeline details of the key mechanisms. In Section 4, we report our experimental results. In Section 5, we describe related work and we conclude in Section 6.

2. Skipper Microarchitecture

Figure 1 illustrates the differences between correct prediction, misprediction, and skipping. Figure 1(a) identifies the control-flow dependent (segments **A** and **B**) and control-flow independent, post-reconvergent (segment **C**) computations in a program segment, as defined in Section 1. Figure 1(b) shows the timelines of correct and incorrect predictions. Correct prediction leads to execution overlap among the instructions before the branch, and the instructions from the predicted path (**A** and **C** segments). A misprediction usually leads to squashing of *all* instructions after the branch, irrespective of whether they are control-flow dependent or independent (both **B** and **C** segments).

Figure 1(b) shows Skipper's timeline. Skipper overlaps the computation before the difficult branch (and resolution of the branch), with the post-reconvergent instructions that are data independent of the skipped instructions (data independent instructions from segment **C**). On resolving the difficult branch, Skipper suspends execution of the post-reconvergent instructions. Skipper then executes the correct path in the skipped computation (segment **A**), allowing the post-reconvergent instructions that are data dependent on the skipped instructions to proceed (rest of segment **C**). After fetching all the skipped instructions till the reconvergence point, Skipper continues with the suspended post-reconvergent computation.

Despite its advantages, skipping is not always beneficial. Skipping branches that would be correctly predicted may cause performance loss, while not skipping branches that would be incorrectly predicted results in lost opportunity. Comparing correct prediction and skipped timelines in Figure 1(b) reveals this point. The performance loss is incurred because conventional superscalars do not delay any of the instructions following a correctly predicted branch, but Skipper unnecessarily delays the skipped instructions and the post-reconvergent data-dependent instructions (from **C**), until the difficult branch is resolved.

2.1. Overview of the Skipper microarchitecture

We describe Skipper based on an out-of-order pipeline using rename tables for register renaming and an instruction window for out of order issue. Skipper employs the JRS scheme [9] to identify the branches that are repeatedly mispredicted by the branch predictor. Basically JRS monitors the prediction accuracy of prior instances of branches

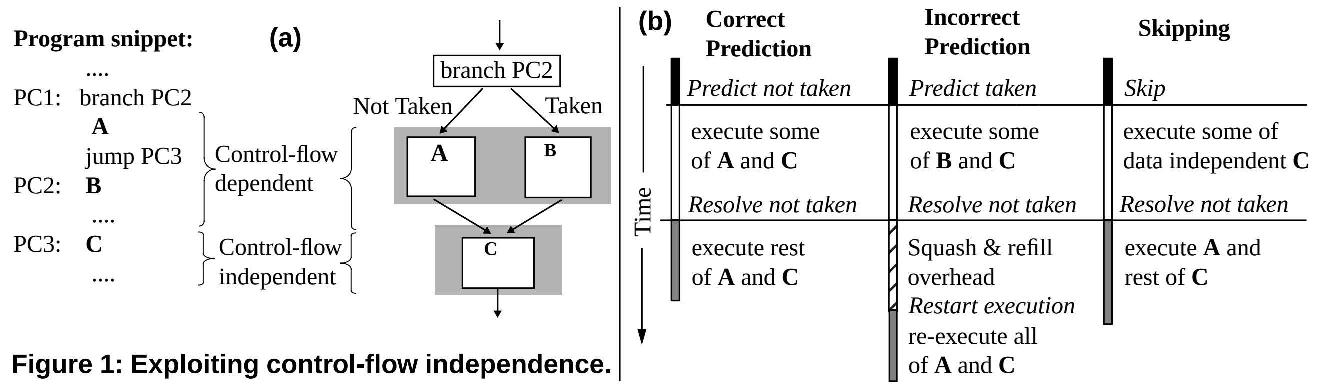


Figure 1: Exploiting control-flow independence.

and isolates branches with low accuracy. Along with the prediction of every branch, JRS determines if the branch should be skipped. If so, Skipper uses a heuristic to determine the branch's reconvergence PC, and maintains the PC in the Skipped Computation Information Table (SCIT). Subsequent instances of the branch obtain the reconvergence PC from the SCIT.

In the following cycles, Skipper fetches the post-reconvergent instructions and places them in the instruction window. Skipper creates a contiguous gap, large enough to hold all the skipped instructions (both taken and not-taken paths), in the instruction window, and places the post-reconvergent instructions after the gap, similar to [21]. Skipper learns the *likely* maximum gap length, also maintained in the SCIT, by counting the number of instructions in the skipped computation (in both the taken and not-taken paths) in prior instances of the branch. After the skipped branch resolves, Skipper fills the gap with the skipped instructions from the correct (taken or not-taken) path. Instructions in the instruction window remain in program order, and Skipper maintains precise interrupts despite out-of-order fetch.

Using the fact that conventional superscalars fetch instructions in program order, the register rename table links register value producers to consumers, and the load/store queue deduces producer-consumer relationships for memory values. Because Skipper fetches the post-reconvergent instructions out of order before the skipped instructions, as such the pipeline cannot establish data dependencies among the skipped instructions and the post-reconvergent instructions. Previous schemes that fetch instructions out of order face similar problems: The Multiscalar architecture uses the compiler to specify register dependencies [3]. The Dynamic Multithreading architecture employs value speculation and intricate recovery [1].

In conventional out-of-order pipelines' rename stage, instructions map their architectural destination register to a new physical register, and place the new, architectural to physical rename map in the master rename table. Out-of-order fetch presents two issues for Skipper's register renaming. First, Skipper has to ensure that the rename maps for the skipped instructions' source registers are not clobbered by the post-reconvergent instructions. Second, Skipper has to ensure that the post-reconvergent consumer instructions obtain the correct rename maps corresponding to the skipped producer instructions. To handle these issues, Skipper learns the set of architectural source registers, the *inputreg set*, and destination registers, the *outputreg set*, for the skipped instructions (both taken and not-taken paths) in prior dynamic instances. The SCIT holds the inputreg and outputreg sets. The outputreg set is similar to Multiscalar's create mask, except Multiscalar uses the compiler to determine this information [23], and Skipper uses hardware.

For the first issue, Skipper copies the rename maps corresponding to the inputreg set from the master rename table to a backup rename table, at the time of skipping a branch. At this point, the master table reflects the register state of the program at the difficult branch. Post-reconvergent instructions modify the master rename table, and not the backup table. Later, when the skipped instructions are fetched, they use the maps in the backup table.

For the second issue, Skipper forces the data-dependent, post-reconvergent instructions to wait till the yet-to-be fetched skipped instructions execute. The outputreg set gives Skipper a priori, albeit approximate, knowledge of the destination registers for which the yet-to-be fetched skipped instruction. At the time of skipping a branch, Skipper *preallocates* and *preassigns* physical registers for the outputreg registers (e.g., map architectural outputreg register R3 to preallocated physical register P103), and marks the physical registers busy. Much like a data-dependent instruction in superscalars, any post-reconvergent instruction that is data dependent on a skipped instruction waits till the corresponding preassigned physical register is ready or bypassed. When the skipped instruction eventually completes execution, its preassigned physical register gets the value, allowing all waiting post-reconvergent instructions to proceed. Data-independent, post-reconvergent instructions proceed without waiting, much as in superscalars.

Because several skipped computations could be in flight, Skipper uses multiple backup and preassign tables, much as superscalars use a backup rename table for each unresolved branch in flight.

For memory dependencies, Skipper faces the same problem of imposing program order in the load/store queue, as in the instruction window. Skipper creates an appropriately-sized gap in the load/store queue in parallel with the instruction window, and maintains the load/store queue gap length information also in the SCIT. Skipper exploits conventional load/store queues' ability to allow loads to proceed without knowing all previous store addresses, letting post-reconvergent loads to proceed even though the skipped stores have not even been fetched. Conventional load/store queues check if later loads complete prematurely before an earlier store to the same address, and enforce store-load program order via squash and rollback. Skipper can avoid such squashes using well-known memory dependence synchronization techniques [16]. Thus, Skipper's loads and stores remain in program order in the load/store queue, despite out-of-order fetch.

3. Supporting control-flow independence

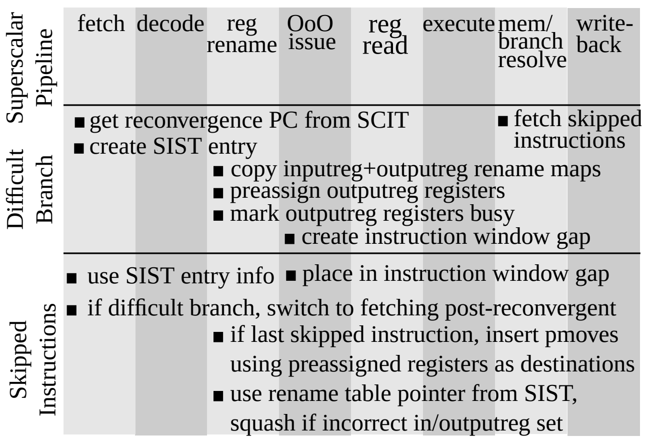
Before we describe the details of how the required information is gathered in the SCIT (Section 3.2), we explain how we use the SCIT information.

3.1. Using the SCIT information

The Skipper pipeline treats instructions that are not skipped as well as branches that are not difficult much like a conventional superscalar pipeline. The cases where Skipper's actions are different from those of a conventional pipeline are (1) when Skipper identifies a branch to be difficult, (2) when Skipper resolves a difficult branch, (3) when Skipper fetches and executes a skipped computation, and (4) when Skipper fetches the last instruction from a skipped computation. The post-reconvergent instructions flow through the pipeline *without* any special actions. Figure 2 shows an out-of-order pipeline extended with Skipper. We do not show post-reconvergent instructions.

3.1.1. Fetching a difficult branch

Using the predicted PC, the front-end of the Skipper pipeline probes the JRS structure and the SCIT, in addition to the usual branch prediction tables. If the JRS structure identifies a branch to be difficult, the fetch stage fetches from the reconvergence PC provided by the SCIT. If the SCIT does not have an entry for this branch or if the instruction window gap length as provided by the SCIT entry is larger than the *gap-length-threshold*, then Skipper defaults to branch prediction, overruling the JRS's recommendation. Gap-length-threshold ensures that Skipper does not create inordinately large gaps in the instruction window, under utilizing the instruction window. Skipper obtains the reconvergence PC in parallel with the fetching of the difficult branch, much like a branch target address from the BTB in conventional pipelines, Thus, Skipper fetches the post-reconvergent instructions in the immediately following cycle after the difficult branch fetch cycle, without inserting any bubbles in the pipeline.



Superscalar Pipeline
fetch
decode
reg rename
OoO issue
reg read
execute
mem/ branch resolve
write-back
Difficult Branch
■ get reconvergence PC from SCIT
■ create SIST entry
■ copy inputreg+outputreg rename maps
■ preassign outputreg registers
■ mark outputreg registers busy
■ create instruction window gap
■ fetch skipped instructions
Skipped Instructions
■ use SIST entry info
■ place in instruction window gap
■ if difficult branch, switch to fetching post-reconvergent
■ if last skipped instruction, insert pmoves using preassigned registers as destinations
■ use rename table pointer from SIST, squash if incorrect in/outputreg set

Figure 2: Skipper pipeline.

Skipper allocates an entry in the Skipped Instruction Status Table (SIST) for every skipped branch to hold information required by various pipeline stages for the skipped computation and the post-reconvergent computation. There could be multiple difficult branches in flight in the pipeline, and the SIST holds an entry for every difficult branch in flight. However, Skipper does not perform nested skipping (i.e., skipping within a skipped computation), and so the SIST entries are in program order. Every instruction carries its SIST entry number so that the instruction can be associated with its SIST entry in later pipeline stages.

Conventional pipelines allocate a history rename table when a branch enters the rename stage. Subsequent instructions copy the previous rename map of their destination registers from the master table to the history table, before placing their new rename map in the master table. This procedure essentially checkpoints the rename maps, allowing fast recovery of the maps on mispredictions. The copying is done at a rate matching the issue width. For instance, in a four-issue machine, the rename table allows, in one cycle, eight reads for the sources, and four reads and eight writes to checkpoint the old maps and update the new maps for the destinations. Rename table bandwidth is a critical resource, and as such the entire table (e.g., for 64 architectural registers and 512 physical registers, the table has 576 bits) *cannot* be backed-up en masse, in one cycle. In [24], this point was noted in the Mapping Synchronization Bus description.

When the difficult branch reaches the rename stage, Skipper modifies the rename tables as per the inputreg and outputreg sets. To that end, Skipper first allocates a *backup rename table* and copies the rename maps for the skipped instructions' inputreg and outputreg sets into the backup table. In Section 2.1, we explained why the backup table holds the inputreg maps, but the reason for copying the outputreg maps will become clear in Section 3.1.3. This copying proceeds at the bandwidth provided by the backup table and may stall the rename stage for as many cycles as needed for the copying.

Skipper then preallocates and preassigns new physical registers to the skipped instructions' outputreg set. Skipper updates the master table with the preassigned physical register rename maps, and marks the preassigned physical registers as busy. Additionally, Skipper allocates another *preassign rename table* and updates the table with the outputreg set's preassigned physical register rename maps. The updating of the master and preassign tables too proceed at the bandwidth provided. Skipper places pointers to the backup and preassign tables in the difficult branch's SIST entry so that when the skipped instructions are fetched, the pipeline knows which rename table to use.

In the out-of-order issue (OoO issue) stage, Skipper uses the difficult branch's instruction window and load/store queue gap length information from the SCIT to create a gap in the instruction window and load/store queue. Skipper puts pointers to the instruction window and load/store queue gaps in the difficult branch's SIST entry so that on fetching the skipped instructions, the pipeline knows where to place them.

3.1.2. Resolving a difficult branch

Till Skipper resolves the difficult branch, execution proceeds with the post-reconvergent instructions much like conventional pipelines, and the pipeline front-end predicts branches. The post-reconvergent instructions modify the master table, as usual. If JRS identifies a subsequent branch to be difficult, Skipper continues at the branch's reconvergence points, allowing multiple skipped branches in flight.

Upon resolving a difficult branch, the fetch stage is diverted to fetch from the correct path of the skipped computation, temporarily suspending fetching from the post-reconvergent computation. Skipper provides the correct branch target to the fetch stage along with the branch's SIST entry number so that the skipped instructions are associated with the correct SIST entry. Skipper holds the PC up to which the post-reconvergent instructions have been fetched in the difficult branch's SIST entry, so that after the skipped instructions are all fetched, the fetch stage can revert back to fetching the post-reconvergent instructions starting from that PC. The skipped instructions from the correct path enter the pipeline starting from the cycle following the branch resolution. This change of fetch stream does not entail any pipeline bubbles because the post-reconvergent instructions flow through the pipeline, as before.

If the post-reconvergent instructions fill up the instruction window (except for the gap) and the front-end pipeline stages from OoO issue all the way back to fetch, Skipper may deadlock. Basically the skipped instructions cannot get into the pipeline even though there are instruction window slots set aside for them. Skipper avoids such deadlocks by squashing the instructions in the stages from OoO issue back to fetch, freeing up the front-end stages so that the skipped instructions can get into the instruction window.

3.1.3. Fetching and executing skipped instructions

The skipped instructions, carrying the SIST entry numbers provided by the difficult branch, pass through the pipeline. Skipper places the skipped instructions in the instruction window, and loads and stores in the load/store queue using the SIST entry's instruction window entry pointer and load/store queue entry pointer, respectively. The skipped instructions use the rename tables identified by the backup and preassign table pointers, stored in the SIST entry. The backup table contains both inputreg and outputreg registers' rename maps corresponding to the register state of the program at the difficult branch.

The skipped instructions use the backup (and not master) table both to get their source rename maps and to put their destination rename maps. If a skipped instruction's source is an inputreg register, the backup table provides the rename map for the register. There are two issues with guaranteeing correctness with regard to the outputreg registers. First, multiple skipped instructions writing to the same architectural destination register pose a problem because Skipper preassigns only one physical register per outputreg register. Second, Skipper needs to identify when it is safe for the dependent post-reconvergent instructions waiting on the preassigned registers to use the values in the registers. Because multiple skipped instructions may write to the same architectural register, it may not be correct to allow a dependent instruction to read the register as soon as a write occurs.

Skipper handles both issues using a simple approach. In the rename stage, the skipped instructions do not use the preassigned physical registers as their destinations. Instead, these instructions obtain newly allocated physical registers. As skipped instructions pass through the rename stage, they update the backup table with the new physical register maps. Subsequent skipped instructions obtain the correct rename maps for their source registers from the backup table. At the end of the skipped computation, Skipper introduces extra physical register move instructions called *pmove*s, similar to [10,20]. Pmoves (described in Section 3.1.4) copy the latest outputreg value from the physical registers given by the backup table maps to the preassigned registers given by the preassign table maps.

Because the outputreg set is an estimate based on previous instances, an outputreg register may not be written by the skipped instructions. In that case, the latest value for an outputreg register comes from an instruction before the difficult branch. It is for this reason Skipper copies the outputreg rename maps into the backup table when the difficult branch is in the rename stage, as mentioned in Section 3.1.1. Consequently, the backup table holds the latest rename map for the outputreg registers irrespective of whether the skipped instructions actually write to the outputreg registers or not, and therefore, the pmoves copy the correct values.

Conventional pipelines free the previous physical register mapped to the same architectural register as the committing instruction's destination. Because Skipper commits instructions in program order, this approach works for Skipper too. Clearly this approach works for all instructions up to the first gap. At the gap, previous physical registers fall into two categories: either they are mapped to outputreg registers or not. Those mapped to outputreg registers are freed by writes in the gap, and the writes' registers are freed by pmoves; if there are no writes, pmoves directly free the previous registers. Those not mapped to outputreg registers are freed by post-gap instructions, as usual.

It is possible that an architecture register not in the outputreg set is written to in the skipped computation. A dependent post-reconvergent instruction may incorrectly use a stale value assuming that the register would not be written by the skipped instructions. A similar situation is possible for the inputreg set, where a skipped instruction needs to read a register not in the inputreg set. These conditions are easily detected in the register rename stage by comparing each skipped instruction's destination (source) register against the outputreg (inputreg) set of the instruction's SIST entry. On detection, Skipper simply squashes all post-reconvergent instructions and triggers recovery of the missing register's rename map, irrespective of whether or not an incorrect value or rename map was used.

While executing the skipped instructions, Skipper predicts the branches within the skipped computation, as usual. Incorrect branch prediction within the skipped computation results in squashing all post-reconvergent instructions, nullifying Skipper's ability to exploit control-flow independence. If JRS identifies a branch within skipped computation to be difficult, Skipper suspends fetching from the branch till the branch is resolved and reverts to fetching from the post-reconvergent stream, using the post reconvergent fetch PC in the SIST entry. While this simple solution further delays the dependent post-reconvergent instructions, it avoids squashing post-reconvergent computation. Another solution is to skip the difficult branches within the skipped computation but such nested skipping may complicate implementation.

Out-of-order fetching may interact with branch prediction unfavorably because speculative update of branch history [8] may be disrupted by the out-of-order fetch stream. Because this is the first paper on this approach, we avoid this issue by assuming that branch prediction updates occur at commit point, although previous results have shown speculative updates to perform better than commit updates.

3.1.4. Last instruction in the skipped computation

Each SIST entry holds the corresponding reconvergence PC to allow the fetch stage to determine when a skipped instruction stream merges with its post-reconvergent computation and stop fetching more instructions from the skipped stream. Every cycle, the fetch stage compares the next fetch PC with the reconvergence PCs held in the SIST entries, and on a match stops fetching from the corresponding skipped computation further. Skipper then inserts the extra pmoves into the instruction window, so that they execute as and when the value for the outputreg registers become available. On execution, the pmoves write to the preassigned physical registers and mark them ready, allowing dependent, post-reconvergent instructions to proceed. The fetch stage then reverts to the post-reconvergent computation by continuing from the PC at which the post-reconvergent stream was left off.

Because the instruction window and load/store queue gap lengths are estimates based on previous instances, it is possible that the gaps in the instruction window and load/store queue fill up before all the skipped instructions are fetched. In that case, Skipper simply squashes all the post-reconvergent instructions to make room for the rest of the skipped instructions to be placed in the instruction window. If the reconvergence PC obtained by the pattern-matching heuristic is incorrect, the effect of this incorrect information is that the instruction window gap fills up before the skipped instruction stream merges with the post-reconvergent computation, causing Skipper to squash all the post-reconvergent instructions starting from the incorrect reconvergence PC.

3.2. Learning the SCIT information

Skipper learns all the required information about the skipped computation from previous instances and deposits them in the SCIT for subsequent instances. The information collected in the SCIT are: identifying which branches are difficult, what the reconvergence PCs are, what the instruction window and load/store queue gap lengths should be, and the skipped instructions' outputreg set.

3.2.1. JRS for identifying the branches to skip

Skipper uses JRS to identify difficult branches by accessing the JRS structures in parallel with every branch prediction. Basically, JRS tracks the number of times a branch is mispredicted using saturated counters, much like branch prediction schemes. The counters count up on incorrect predictions and count down on correct predictions. Depending upon the desired accuracy and coverage rates, JRS chooses appropriate values for both the up/down rates and the count threshold above which a branch is deemed difficult. Even if a branch is deemed difficult, JRS and branch prediction continue to make predictions and update the tables. If a branch is repeatedly predicted correctly, JRS stops marking the branch as difficult [12].

The key aspects of JRS relevant to Skipper is that skipping branches that would be correctly predicted may cause performance loss, while not skipping branches that would be incorrectly predicted results in lost opportunity. The performance loss is incurred because conventional superscalars do not delay any of the instructions following a correctly predicted branch, but Skipper unnecessarily delays the skipped instructions and the post-reconvergent data-dependent instructions following the branch, until the branch is resolved. Thus, it is a trade-off between JRS's coverage and accuracy, and while lower coverage means lost opportunity, lower accuracy may mean performance loss.

3.2.2. Heuristic for identifying the reconvergence point

For if-then-else constructs in high-level languages, the compiler typically generates a branch to determine whether the if clause or the else clause is to be executed. The compiler also generates a jump to the reconvergence PC at the end of the if clause, to elide the else clause. Therefore, the reconvergence PC can be determined if the jump is located. The target of the branch is the start of the else clause and the jump instruction is located immediately before the branch target. For example, in Figure 1(a), the branch target, *PC2*, is at the start of segment **B**, which is the else clause. The *jump PC3* immediately before PC2 jumps to *PC3*, which is the reconvergence PC. Accordingly, Skipper computes the target of the difficult branch and uses the PC immediately before the branch target to probe the i-cache and inspects the instruction there. If the instruction is a jump instruction, then the target of the jump instruction is the reconvergence PC.

If the instruction at the PC immediately before the difficult branch target is not a jump, then Skipper assumes that the difficult branch is from an if-then construct, instead of an if-then-else construct. For if-then constructs, the compiler generates a branch to elide the if-clause instructions if the condition is false, and the branch target is the reconvergence PC. If a difficult branch is a backwards branch (indicated by a negative offset), neither of the above heuristics work. Conceptually, the difficult branch being a loop branch indicates that number of loop iterations is hard to predict, and accordingly, Skipper designates the reconvergent point to be the exit out of the loop (i.e., the PC immediately after the loop branch). Unlike previous work [18], we do not include the return PC of a function as the reconvergence PC of all branches within the function body because of gap-length-threshold constraints.

Using only one probe into the i-cache, Skipper's heuristic determines the reconvergence PC. Because the probe is done only for difficult branches and not all branches, and that too only if the SCIT does not have the reconvergence PC, this probe does not degrade i-cache bandwidth. Once the reconvergence PC of a branch is recorded in the SCIT, the heuristic is not used until the SCIT replaces the branch's entry due to capacity or conflict issues. Because Skipper obtains the reconvergence PC most of the time from the SCIT and not the heuristic, computing the difficult branch's target for the heuristic can be slow. Consequently, this computation is done over many pipeline stages instead of just decode, without affecting the cycle time.

There are compiler optimizations that may confuse the heuristic. For instance, in code layout optimization to improve i-cache performance, the compiler moves infrequent control-flow paths away from the sequential stream. Such code motion changes the code pattern and renders the heuristic ineffective. However, this optimization may be applied to only those branches that are biased towards one of the two paths, otherwise one path would not be more frequent than the other. So, such branches may not difficult to predict and may not need to be skipped. Other optimizations may cause exceeding of the gap-length-threshold. An example is tail duplication of the post-reconvergent code into the if and else paths, increasing the gap length.

3.2.3. Estimating the gap length

Once JRS identifies a difficult branch and the reconvergence heuristic determines the reconvergence PC, Skipper collects the instruction window and load/store queue gap length information from subsequent instances of the branch. Upon committing the difficult branch, Skipper creates a valid entry in the Gap Information Learning Buffer (GILB), and places the reconvergence PC in the GILB entry. From the difficult branch onwards, every committing instruction increments the instruction window gap length count of *all* valid GILB entries, because each valid GILB entry represents a distinct difficult branch whose reconvergence PC has not been committed. Also, Skipper matches the PC of the committing instruction against the reconvergence PCs of all the valid GILB entries. A match indicates that the corresponding difficult branch's reconvergence PC has been reached. Skipper transfers the information in the GILB entries to the SCIT, and relinquishes the GILB entry.

To keep the SCIT information as accurate as possible, Skipper continues to collect the information in subsequent instances of the difficult branch, irrespective of whether the branch is predicted or skipped. If the instruction window (or load/store queue) gap length count in any later instance is larger than the length recorded in the SCIT, Skipper updates the SCIT entry with the larger count. If the count is smaller, it is discarded. This repeated updating of the maximum length helps Skipper account for different control-flow path lengths within the skipped computation.

If Skipper does not track the maximum length, Skipper would essentially have to predict the skipped computation's path length to estimate the gap length. Predicting the path length may indirectly lead to predicting the difficult branch, defeating Skipper's purpose. Because the maximum length is longer than all but the longest path within the skipped computation, Skipper is conservative in setting aside the instruction window gap. Some of the instruction window slots remain empty if the actual path within the skipped computation is not the longest. This conservative choice is better than predicting the difficult branch because the number of wasted slots is still much smaller than the slots spent on numerous incorrect instructions (including not only incorrect control-flow dependent instructions, but also control-flow independent instructions which are data dependent on the incorrect control-flow dependent instructions) in out-of-order superscalars and other approaches (Section 1).

3.2.4. Determining outputreg and inputreg set

Along with the gap lengths, the GILB also tracks the outputreg and inputreg set of the skipped computation using a bit-vector field in the GILB entries. From the difficult branch onwards, every committing instruction's destination (source) register is added to the outputreg (inputreg) bit-vector of *all* valid GILB entries. Because Skipper collects outputreg and inputreg information at commit point, incorrect predictions within the skipped computation do not adversely affect the accuracy of the information. When the gap length information is transferred to the SCIT from the GILB, the GILB entry's bit-vectors are bit-wise ORed with the SCIT entry's corresponding bit-vectors. The ORing accounts for different outputreg and inputreg sets along different control-flow paths within the skipped computation. The outputreg and inputreg set are conservative union over all control-flow paths within the skipped computation. Not considering all the paths would cause Skipper to predicting the difficult branch indirectly, as argued above.

4. Experimental Results

We modified the Simplescalar2.0 simulator [4] to model Skipper. Table 1 shows the base system configuration parameters used throughout the experiments, unless specified otherwise.We assume a hybrid predictor and 9-cycle misprediction penalty. We assume generous branch prediction tables each of which has 8K entries to allow as high a prediction accuracy as possible, but a modest SCIT size of about 3KB. We use a bimodal JRS with 4K, 4-bit entries for a total of 2KB. We model the return address stack (RAS), and account for RAS and BTB mispredictions which are not addressed by Skipper.

We accurately model the extra rename bandwidth to handle inputreg and outputreg sets, and the extra pmoves at the end of skipped computation. We do not include any memory dependence synchronization mechanisms, but account for memory dependence squashes. Because Skipper's key advantage is in conserving i-cache bandwidth, we carefully model i-cache bandwidth. Table 2 presents the SPECint95 benchmarks and their inputs used in this study. We run the benchmarks to completion.

4.1. Performance of Skipper

In this section, we present the base performance of Skipper, compared to an out-of-order superscalar with branch prediction. In the two left bars, we vary the instruction window size from 128 to 256 entries. The speedups are normalized against an out-of-order superscalar with the same instruction window size. The numbers above the bars are the speedups for a 128-entry superscalar with perfect branch prediction, to serve as an idealized reference. Because Skipper uses extra storage for SCIT and JRS (about 6KB) compared to a superscalar, we also show Skipper's speedups normalized with respect to a "large superscalar" using an extra 6KB in larger prediction tables (16K entries each table, total size 12KB) in the two right bars. For both Skipper and superscalar, we use one port for the 128-entry instruction window and two ports (and double front-end width for decode and rename stages) for the 256-entry instruction window. Each cycle only one block can be fetched through one port and the entire block may not be useful due to branches and jumps within the block. For the 256-entry Skipper and superscalar, we assume aggressive front-ends that can obtain two fetch PCs from the branch predictor and use both the ports for fetching.

Table 1: Hardware parameters for base systems.

Processor	8-way issue,128-entry window, 43-entry load/store queue, (9 cycle branch penalty)
Branch prediction	8k/8k/8k hybrid, 4k 4bit JRS, gap length threshold 48, 128-entry RAS, 4-way 4K BTB
Caches	64KB 2-way 2-cycle I/D L1, 2MB 8-way 12-cycle L2, both lockup free and pipelined
Main memory	Infinite capacity, 100 cycle latency; split transaction, 32-byte wide bus
SCIT	3KB: 128 entries each 199-bit wide, 4-way (24-bit tag, 32-bit reconvergence PC, 67-bit outputreg and inputreg, 6-bit instruction window gap length, 6-bit load/store queue gap length)
SIST	864 bytes: 36 entries each 192-bit wide (fetch PC 32 bits, 7-bit pointers for instruction window, load/store queue and RAS, 5-bit rename table pointer, 67-bit outputreg and inputreg sets
GILB	896 bytes: 36 entries each similar to SCIT entry

Table 2: Benchmarks and inputs.

	Input	# insts		Input	# insts
cc1	cccp.i (test)	$1.3*10^9$	compress	train	$36*10^6$
go	2stone9	$548*10^6$	ijpeg	vigo	$1.5*10^9$
li	test.lsp	$957*10^6$	m88ksim	ctl.in(test)	$490*10^6$
perl	jumble	$2.4*10^9$	vortex	train	$2.5*10^9$

From the first two bars in Figure 3, we can see that for a 128-entry instruction window, Skipper achieves a wide range of speedups up to 15% for *ijpeg*, 14% for *li*, *9% for go*, and 8% for *m88ksim*, all the way down to small slowdowns for *perl* and *compress*. These speedups indicate that

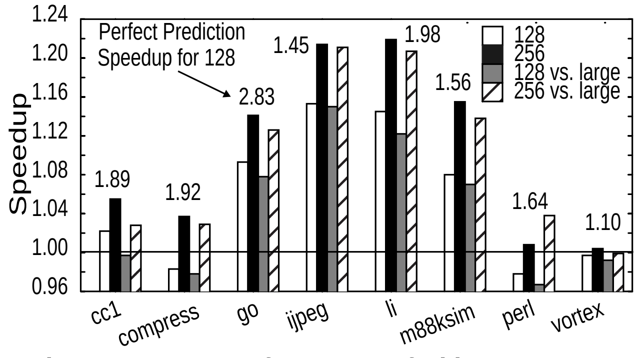


Figure 3: Base performance of Skipper.

Skipper successfully skips around difficult branches and overlaps branch resolution with control-flow independent instructions. With a 256-entry instruction window, Skipper achieves higher speedups, resulting in up to 22% speedups for *ijpeg* and *li,* 16% for m88ksim, and small speedups for *compress*. Increasing the instruction window size improves speedups because Skipper uses the extra entries better than a conventional superscalar. While Skipper brings more useful instructions into the extra entries in the instruction window, conventional superscalar is limited by mispredictions and squanders the extra entries on incorrect instructions.

The two right bars show Skipper's speedups normalized against the "large superscalar". Comparing the left bars with the right bars for the same instruction window size, we see that the change in Skipper's speedups is less than 3% in all cases. These results indicate that the extra prediction storage does not give superscalar much performance advantage and is better used by Skipper.

4.1.1. Effectiveness of Skipper's mechanisms

Skipper's speedups widely vary across benchmarks and are still far lower than those for perfect branch prediction; the measurements in Table 3 explain the reasons. *JRS coverage* (related to the metrics in [6]) is the ratio of the number of branches JRS identifies as difficult to the total number of mispredicted branches. *Heuristic accuracy* is the ratio of the number of branches with correctly-determined reconvergence PC to the total number of branches with reconvergence PC within the gap-length-threshold. *Actual coverage* is the ratio of the number of actually skipped branches to the total number of mispredicted branches. Actual coverage measures the opportunity exploited by Skipper. JRS coverage attenuates to actual coverage due to both mispredicted branches having reconvergence PCs beyond the gap-length-threshold and the heuristic determining reconvergence PCs incorrectly.

Overshoot is the ratio of the number of skipped branches which would have been correctly predicted in a superscalar to the total number of branches. Overshoot measures unnecessary stalling. *Reconvergence accuracy* is the ratio of the number of successfully skipped branches to the number of actually skipped branches. A successfully skipped branch is one for which the reconvergence PC is reached within the instruction window gap, and there are no squashes due to in-gap branch mispredictions (Section 3.1.3), incorrect outputreg set, or skipped stores (Section 3.1.1). Reconvergence accuracy measures the accuracy of SCIT information learnt by Skipper. Skipper's misprediction rate is the ratio of the number of incorrectly predicted and unsuccessfully skipped branches to the total number of branches.

We see that overshoot is mostly less than about 11% and reconvergence accuracy is usually higher than 95%, but actual coverage is low. While JRS coverage is about 78%-98%, actual coverage falls within a mere 17-58%. We experimented with JRS's parameters but could not obtain significantly better JRS coverage for *cc1*, *ijpeg*, *m88ksim*, and *vortex*. Actual coverage can fall far below JRS coverage due to either poor heuristic accuracy or reconvergence PCs being farther than the gap-length-threshold. Heuristic accuracy is 75%-100%, which is too high to degrade actual coverage by a large margin, implying that gap-length-threshold prevents a large fraction of difficult branches from being skipped. We vary gap-length-threshold in Section 4.4, but found that many difficult branches have far away reconvergence points (more than 200 instructions), requiring inordinate gap-length-threshold values. Missed opportunity due

Table 3: Measurements of Skipper's mechanism. **Table 4: Gap characteristics.**

Bench-marks	JRS coverage %	Heuristic accuracy %	Actual coverage %	Overshoot %	Reconver-gence accuracy %	Skipper's mispredict rate %	Superscalar mispredict rate %	#gaps	#in	#out	#instr	#slot
cc1	92	75	19	7	92	8	10	1.4	6	4	7	14
compress	98	100	25	9	100	8	12	1.5	4	3	4	10
go	98	87	20	11	89	16	24	1.4	8	5	10	21
ijpeg	90	96	58	8	98	3	9	2.0	6	4	5	13
li	96	77	17	6	100	4	8	1.2	5	3	9	16
m88ksim	78	90	32	11	99	2	4	2.1	4	2	5	9
perl	94	98	16	2	100	3	4	1.3	5	3	4	8
vortex	88	79	17	2	98	1	1	1.0	5	3	8	13

to low actual coverage is the key reason for Skipper lagging far behind perfect prediction.

cc1 and *go* incur many mispredictions both within and outside skipped computations (gaps). These mispredictions are not caught by Skipper due to its low coverage and cause squashing of post-reconvergent computation, nullifying Skipper's advantage. *compress* runs out of instruction window slots for a 128-entry instruction window, alleviated only slightly by a 256-entry instruction window. In addition to mispredictions, *go* also incurs many memory dependence squashes (*go* is the only benchmark with this problem). *ijpeg* and *m88ksim* have higher coverages than the rest, translating to higher performance. In *li*, Skipper skips *entire* short unpredictable loops (dynamic instructions in all iterations less than 10). Because loop back branches within skipped loops in *li* are not predicted but suspended till resolution, *li* avoids many mispredictions, achieving high speedups. *li*'s coverage is small because the coverage numbers do not include such suspended branches which are not mispredicted but not skipped either. *Perl* has many non-return, indirect jump mispredictions both within and outside skipped computations, which have the same effect as *cc1*'s mispredictions. *Vortex*'s prediction accuracy is high, leaving little opportunity for Skipper.

4.1.2. Characteristics of skipped computations

Table 4 shows the average number of actually skipped branches in flight ("#gap" column), inputreg and outputreg registers per difficult branch ("#in" and "#out" columns), dynamic instructions per skipped computation ("#instr" column), and the average instruction window gap length, not including pmoves ("#slot" column). The benchmarks have less than two difficult branches in flight, implying that only two skipped computations need to be tracked in the GILB and SIST. The number of skipped instructions ranges between four and ten, even though the gap-length-threshold is 48, implying that the threshold is not hit often. The difference between the gap lengths and the number of skipped instructions is about seven, implying that Skipper wastes only a few instruction window slots. The number of outputreg registers being about four means that Skipper inserts around four extra pmoves, which could execute together in one cycle on a 4-way issue machine. The number of inputreg and outputreg registers together is about ten implying that Skipper needs to handle only ten rename maps per skipped branch. In comparison, rename tables in a 4-way issue machine handle 12 registers (8 sources and 4 destinations) every cycle, suggesting that Skipper incurs low rename bandwidth overhead.

4.2. Comparison between Skipper and Polypath

In this section, we compare Skipper against the previously-proposed Polypath architecture [12, 11]. We vary the configuration from 128 instruction window entries with one i-cache port (two left bars) to 256 entries with two i-cache ports (two right bars). The speedups are all normalized to a superscalar with equal instruction window size and equal number of i-cache ports.

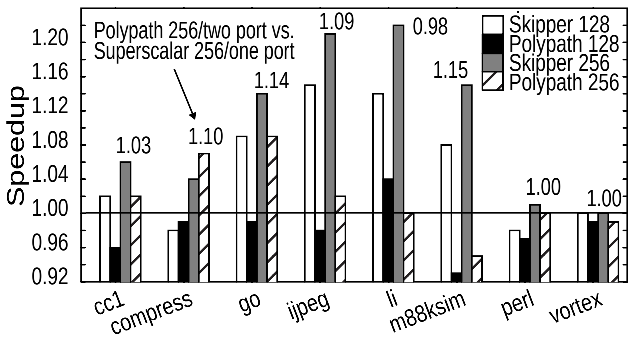


Figure 4: Comparison of Skipper and Polypath.

For Polypath, we use the fetch policy and JRS parameters recommended in [12,11]. Our model of Polypath is different than those in [12,11] in two ways, which affect its speedups. First, the Polypath papers compare a Polypath system using two i-cache ports and double pipeline width for decode and rename stages, with a superscalar using one i-cache port. However, such a comparison fails to isolate the impact of the architecture from the impact of the fetch bandwidth. Therefore, we assume exactly equal fetch bandwidth for Skipper, Polypath, and superscalar. Second, the Polypath papers do not charge any extra cycles to copy the entire rename table (the equivalent of inputreg set) needed to execute both paths of difficult branches. Because of the arguments given in [24] and in Section 3.1.1, we charge cycles for this copying as per the bandwidth of the rename tables. This charging is done for both Skipper and Polypath.

As before, for the 256-entry case, we assume an aggressive superscalar that can obtain two fetch PCs from the predictor and use both the i-cache ports. If there are no difficult branches in flight, both Skipper and Polypath use the ports exactly the way superscalar does. Polypath fetches both paths of difficult branches. Skipper fetches the post-reconvergent stream and the control-flow dependent instructions if there are any resolved skipped branches, and defaults to superscalar mode, if none of the skipped branches are resolved. We also compare a two-port Polypath without charging cycles for rename table copying, against a one-port superscalar and show the numbers above the bars.

From the left two bars in Figure 4, we see that for the 128-entry case, Skipper outperforms Polypath significantly for *ijpeg*, *m88ksim*, *li*, and go, and modestly or not at all for the other benchmarks. Polypath achieves no speedups mainly because with only one i-cache port, there is not enough bandwidth to fetch down both taken and not-taken paths on difficult branches. This experiment clearly shows that Skipper achieves speedups because of much more efficient use of i-cache bandwidth than Polypath. From the two right bars, we see that for the 256-entry, two i-cache port case, Skipper outperforms Polypath significantly for go, *ijpeg*, *m88ksim*, and *li*, similar to the 128-entry case. In the case of *compress*, Polypath performs better than Skipper by 3%. Further investigation reveals that *compress* has dense data dependencies, disallowing any overlap of post-reconvergent instructions. Because Polypath executes the skipped instructions without any delay unlike Skipper, compress benefits from Polypath.

Compared to the 128-entry, one-port case, Skipper achieves even higher speedups using 256 entries and two ports, with the exception of *vortex*, indicating that Skipper can better use higher i-cache bandwidths than a superscalar. Also, a two-port Polypath with no rename table copy overhead achieves speedups compared to a one-port superscalar, as shown in previous papers [12,11].

4.3. Misprediction Penalty

To see the effect of deepening pipelines, we varied misprediction penalty as 6, 9, and 12 cycles in Figure 5. On one hand, a longer misprediction penalty gives Skipper the opportunity to achieve higher speedups by eliminating the more-expensive mispredictions. On the other hand, a longer misprediction penalty forces Skipper to find more data independent, post-reconvergent instructions to execute before the difficult branch can fill the pipeline with the correct control-flow dependent instructions. Thus, it is a conflict between opportunity and data independence. We see two trends in speedups on increasing penalty: One in which Skipper's speedups for *ijpeg*, *li*, and *compress* indicating that opportunity overcomes dependencies in these benchmarks. And the other in which Skipper's speedups for the rest of the benchmarks reduce due to dependencies offsetting opportunity. Skipper's low coverage restricts opportunity to avoid mispredictions.

4.4. Effect of gap-length-threshold

Because the analysis in Section 4.1.1 indicates that gap-length-threshold impacts coverage, we varied the gap-length-threshold as 24, 48, and 72. While a larger threshold allows better actual coverage, larger threshold also allows branches with larger gap lengths to be skipped, incurring wasted instruction window and load/store queue slots. Thus, it is a trade-off between coverage and instruction window utilization. Increasing the threshold from 24 to 48 improves coverage for all the benchmarks by about 1%-6% reaching the values shown in Table 3. We found that except for *go*, the rest of the benchmarks are not affected by increasing the gap-length-threshold beyond 48. Increasing the threshold from 48 to 72, go's speedup improves from 8% to about 11%. This experiment shows that Skipper's actual coverage is limited not by the threshold setting but by long gaps inherent in programs.

We also varied the SCIT size as 32, 128, and 512 entries

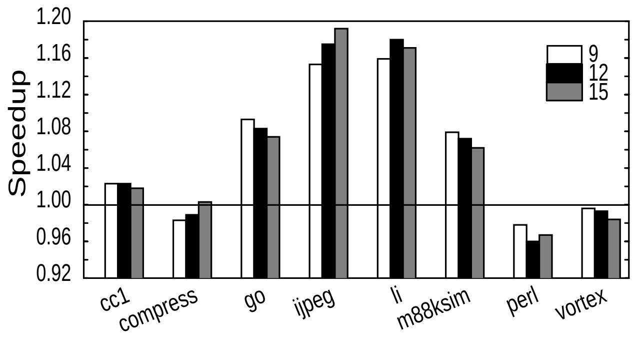


Figure 5: Effect of misprediction penalty.

and found that increasing the SCIT size beyond 128 entries does not improve speedups. Because the base prediction accuracy is high, only a few (static) branches are identified as difficult and they fit within 128 entries.

5. Related work

There have been several results on the potential of exploiting control-flow independence [17]. Many previous ideas to handle difficult branches, amount to executing both the taken and not-taken paths conditioned by such a branch, using varying degrees of ISA support for predication. Proposals such as Multipath [24], Polypath [12,11], dual pipelines [13] and instruction windows [5], and Dynamic Hammock Predication [10] explicitly follow this approach. ISA support for predicated execution removes difficult branches, but at the cost of executing instructions from both the taken and not-taken paths [14,2].

Researchers [17] have proposed selective recovery of control-flow independent instructions after a misprediction, but they point out in a later paper that the scheme is hard to implement [18]. Selective squashing may require expanding/contracting the instruction window at multiple, arbitrary points because the incorrect and correct path instructions are intertwined in the instruction window. Out-of-order pipelines usually track data dependencies through register rename map tables at the granularity of a block of instructions (typically between successive branches), and not individual instructions. This coarse granularity reduces the number of map tables and makes misprediction handling fast and efficient, but disallows fast extraction of selective information about individual instructions. For a realistic selective recovery scheme [18], they propose using Trace processors' hierarchical organization, a solution not applicable to superscalars. (Although Pentium IV has a trace cache, it is not a trace processor.) Instruction reuse [19] is a general technique which can quickly recover control-flow independent instructions after a misprediction. But instruction reuse also squashes all instructions following a misprediction. Multiscalar [20] and Dynamic Mutithreading [1] use hardware or compiler to demarcate threads, which may choose control-flow independent threads to shield intra-thread mispredictions from squashing other threads.

6. Conclusions

Skipper exploits control-flow independence by skipping over control-flow dependent computation of frequently mispredicted branches, in the context of a superscalar pipeline. Skipper fetches the skipped control-flow dependent instructions after the post-reconvergent instructions, out of program order. We describe key mechanisms to implement Skipper without unduly complicating the pipeline despite out-of-order fetch, including (1) identifying difficult branches using the previously-proposed JRS scheme, (2) determining the difficult branch's reconvergence point without scanning, (3) handling out-of-order fetching of the skipped instructions but maintaining program order in the instruction window, and (4) handling data dependencies among the skipped instructions and the yet-to-be fetched post-reconvergent instructions using the existing register rename tables and load/store queue. SPECint95 simulations show that Skipper performs 10% and 8% better than superscalar and the previously-proposed Polypath, respectively, when all three microarchitectures use a 256-entry instruction window and two i-cache ports.

Acknowledgements

This research was partly funded by the NSF CAREER award number 9875960-CCR.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual international symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei Hwu. Integrated predicated and speculative execution in the impact epic architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 181–190, Nov. 1994.

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

[5] Y. Chou, J. Fung, and J. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *International Conference on SuperComputing*, June 1999.

[6] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–131, June 1998.

[8] E. Hao, P.-Y. Chang, and Y. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 228–232, Nov. 1994.

[9] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th annual international symposium on Microarchitecture*, pages 142–152, 1996.

[10] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[11] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32th annual international symposium on Microarchitecture*, pages 38–47, 1999.

[12] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–259, June 1998.

[13] M. J. Knieser and C. A. Papachristou. Y-pipe: A conditional branching scheme without pipeline delays. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 125–128, 1992.

[14] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[15] S. McFarling. Combining branch predictors. Technical Report TR-36, DEC-WRL, June 1993.

[16] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[17] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *5th international symposium on High Performance Computer Architecture*, 1999.

[18] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32th annual international symposium on Microarchitecture*, pages 4–15, 1999.

[19] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[20] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[21] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th annual international symposium on Microarchitecture*, pages 34–43, 1997.

[23] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st annual international symposium on Microarchitecture*, pages 81–92, Nov. 1998.

[24] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[25] T. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.