Dual Path Instruction Processing

Juan L. Aragón[1], José González[1,ϕ], Antonio González[2,ϕ] and James E. Smith[3]

[1]Dept. Ing. y Tecnolog. de Comp.
Universidad de Murcia
30071 Murcia (Spain)

[2]Dept. d'Arquitectura de Comp.
Universitat Politècnica de Catalunya
08034 Barcelona (Spain)

[3]Dept. Electrical and Computing Eng.
University of Wisconsin-Madison
Madison, WI 53706

{jlaragon,joseg}@ditec.um.es

antonio@ac.upc.es

jes@ece.wisc.edu

ABSTRACT

The reasons for performance losses due to conditional branch mispredictions are first studied. Branch misprediction penalties are broken into three categories: pipeline-fill penalty, window-fill penalty, and serialization penalty. The first and third of these produce most of the performance loss, but the second is also significant. Previously proposed dual (or multi) path execution methods attempt to reduce all three penalties, but these methods are also quite complex. Most of the complexity is caused by simultaneously executing instructions from multiple paths.

A good engineering compromise is to avoid the complexity of multiple path execution by focusing on methods that reduce only the pipeline and window re-fill penalties. *Dual Path Instruction Processing* (*DPIP*) is proposed as a simple mechanism that fetches, decodes, and renames, *but does not execute,* instructions from the alternative path for low confidence predicted branches at the same time as the predicted path is being executed. All the stages of the pipeline front-end are hidden once the misprediction is detected. This method thus targets the pipeline-fill penalty and is shown to achieve a good trade-off between performance and complexity. To reduce the window-fill penalty, we further propose the addition of a pre-scheduling engine that schedules instructions from the alternative path in an estimated execution order. Thus, after a misprediction, a high number of instructions from the alternate path can be immediately issued to execution, achieving an effect similar to very fast re-filling of the window. Performance evaluation of *DPIP* in a 14-stage superscalar processor (like IBM Power 4) shows an average IPC improvement of up to 10% for the bzip2 benchmark, and an average of 8% for ten benchmarks from the SPECint95 and SPECint2000 suites.

Categories and Subject Descriptors

C.1.1 [Processor Architectures]: Single Data Stream Architectures.

General Terms

Performance, Design.

ICS'02, June 22-26, 2002, New York, New York, USA.

Keywords

Branch misprediction penalty, dual path processing, confidence estimation, pre-scheduling.

1. INTRODUCTION

Branch mispredictions are well known as a major cause of performance loss in high performance superscalar processors. The conventional way of mitigating this performance loss is to increase the branch prediction accuracy [5][22][24]. To accomplish this, the proposed mechanisms usually increase predictor complexity, which may adversely affect the latency of prediction [15]. Furthermore, many current processors have designs targeted at very high clock frequencies which lead to longer pipelines (e.g. 14 stages in the IBM Power 4 [19] and 20 stages in the Pentium 4 [8]). In such processors, the branch misprediction problem becomes even more crucial because branches take longer to be resolved and instructions take longer to reach execution after a misprediction.

A more radical approach to reducing branch misprediction penalty relies on executing multiple program paths simultaneously [13] [17][26][27]. Branch predictions are assigned a confidence level, and for low confidence predictions, instructions from both paths are decoded, issued, and executed. When the branch outcome is eventually determined, instructions from the wrong path are selectively flushed. A major problem with these techniques is the complexity introduced in the processor design. The size of the physical register file and instruction window must be increased, and selective flushing adds complexity to control. Furthermore, higher execution bandwidth may require more functional units. This may affect not only the cycle time but also the power dissipation because of the execution of more un-committed instructions.

The research reported here is focused on reducing the branch misprediction penalty while striking a balance between complexity, cost, and performance. First, we study the sources of the branch misprediction penalty. Then, we propose a simple scheme that fetches, decodes and renames, *but does not execute*, instructions from the alternative path of low confidence branch predictions while the predicted path is processed as usual. Consequently, instructions from the alternative path can be issued to execution immediately after a misprediction is detected. Finally, we consider an additional mechanism, a pre-scheduling

---

[ϕ] Currently at Intel Barcelona Research Center, Universitat Politècnica de Catalunya, Intel Labs, Barcelona.

engine [4][23], that pre-schedules instructions from the alternative path in a simple way. Thus, instructions from the alternative path can be issued to execution in an estimated execution order, achieving a similar effect to instantaneously filling the instruction window immediately after a misprediction.

The rest of this paper is organized as follows. Section 2 presents the motivation of our proposal and the related work. Section 3 analyzes the sources of performance degradation due to branch mispredictions. The proposed *DPIP* architecture model is described in section 4. Section 5 explains the simulation methodology and analyzes the performance of the *DPIP*. Finally, section 6 summarizes the main conclusions of this work.

2. BACKGROUND AND RELATED WORK

There are two major ways of reducing the performance degradation caused by branch mispredictions. One way is to improve branch prediction accuracy. A very large body of research has been targeted at this solution [5][10][20][21][22][28]. A less-studied approach is to minimize performance degradation by fetching and/or executing multiple paths following a conditional branch. The IBM 3168 and 3033 mainframes could fetch instructions from both paths, though only one path could be decoded and executed [6]. The MIPS R10000 processor has a one cycle delay to decode and predict branches. During this cycle, the processor fetches instructions from the fall-through path. These instructions are stored in a *Resume Cache* [12] partially decoded if the branch is predicted taken. If the prediction is eventually found to be incorrect, the sequential instructions are quickly recovered from the *Resume Cache*.

In [2], the *Misprediction Recovery Cache* (*MRC*) is proposed as a special-purpose cache for an in-order superscalar processor. It is evaluated for an in-order CISC pipeline; thus, instructions are pre-decoded into micro-operations and stored in program order in the cache without register renaming. After a branch misprediction, the *MRC* is probed for a valid subsequent trace. If it is found, the trace is used to feed the execution stages of the pipeline trying to perform a quick recovery.

In [7][9], dual Fetch/Decode mechanisms were evaluated for very simple pipelined processors. A more aggressive approach is proposed in [26], where *Disjoint Eager Execution* (*DEE*) follows a limited number of paths following conditional branches: those more likely to be taken. *DEE* is a limited form of *Eager Execution* (*EE*) that performs better than *EE* when resources are constrained, and helps to speed up the execution of loops. The proposed architecture is implemented on the complex LEVO machine, which uses a fixed-size static instruction window. It can process up to 100 simultaneous branch paths (i.e. the dynamic code between two branches) and executes and resolves up to 256 branches per cycle, something infeasible in current and near future superscalar processors.

In [13], *Selective Dual Path Execution* (*SDPE*) selectively forks a second path, executing instructions from both the predicted and the alternative paths. Decisions about which branches should be forked are based on branch confidence estimation [14]. The proposed implementation duplicates the fetch stage in order to fetch two basic blocks per cycle, one from each path. To support this, the I-cache, BTB, branch predictor, and confidence estimator must be dual ported. Decoding is alternatively performed for both paths in order to simplify the renaming logic, although it requires two *Register Mapping Tables*. In [17], *Selective Eager Execution* (*SEE*) and the *PolyPath* architecture are proposed, generalizing *SDPE* by executing instructions from a limited number of paths. The implemented *PolyPath* architecture can manage up to 8 active paths. In order accomplish this, it can fetch 8 instructions from up to 8 different paths each cycle. Such fetch mechanism puts high pressure on the fetch bandwidth. *Context* (*CTX*) *Tags* are used to identify instructions from different paths, to allow selective flushing of those instructions belonging to the wrong path and their descendents. In [18], several instruction fetch mechanisms are evaluated, studying the effect of fetch bandwidth limitations in multiple path execution processors. *Threaded Multi-Path Execution* (*TME*) [27], uses a *Simultaneous Multithreading* (SMT) architecture [25] and has the ability to execute multiple predicted paths, multiple alternate paths, or both simultaneously. Primary-path branches forking policy is based on branch confidence estimation.

In general, multiple path execution schemes increase the complexity of the processor: bigger register files are needed to store the results of the different instructions. In addition, the size of the instruction window is increased, as well as the size of the reorder buffer. Resource contention may be one of the bigger problems of these architectures, because instructions from a wrong forked path may prevent those of the correct path from being executed. In order to avoid this, resources must be increased: memory ports, functional units, etc. This may affect not only the cycle time but also the power dissipation and energy consumption because many more processor resources are being actively used.

3. BRANCH MISPREDICTION PENALTY ANALYSIS

This section analyzes the sources of performance loss due to branch mispredictions in order to gain some understanding of the performance gap between perfect branch prediction and real branch prediction. We will refer to this overall performance degradation as the *Branch Misprediction Penalty*.

The overall *branch misprediction penalty* can be divided into three components:

1. *Pipeline-fill penalty*. This is the delay between the time the branch misprediction is discovered and the first instruction from the correct path is fetched, decoded, renamed, and placed in the instruction issue window. This penalty is largely a function of the instruction decode pipeline length, but also includes the time required for recovery actions such as flushing the pipeline and restoring the state of the processor.

2. *Window-fill penalty*. This is the performance loss due to the fact that the instruction window is nearly empty for several cycles after the first group of instructions from the correct path has been dispatched. When instructions first begin to re-fill the window, the ILP that can be exploited is first rather small, and begins growing as instruction fetch “catches up”.

3. *Serialization penalty*. A mispredicted branch essentially forces a serialization point in instruction execution. That is,

correct path instructions following the branch cannot be scheduled ahead of the branch (as would be the case if the branch were predicted correctly). In other words, even if there were no *pipeline-fill penalty* or *window-fill penalty*, there would still be a penalty because the correct path instructions would have to wait for the mispredicted branch to be resolved before executing.

To assess each component's contribution to the overall *branch misprediction penalty*, we simulated and measured the IPC for the following scenarios:

a) Real branch prediction using a recovery penalty of *n* cycles. We consider three total pipeline lengths (from fetch to commit) of 6, 10 and 14 stages. In all cases the difference in pipeline length is entirely in the number of stages of the in-order pipeline front-end (fetch/decode)[1].

b) Real branch prediction using a recovery penalty of 0 cycles. After the detection of the misprediction, instructions from the correct path are fetched, decoded, renamed, and placed in the instruction window.

c) The instruction window is filled instantaneously with instructions from the correct path after a misprediction is discovered. The fill process is stopped when the first I-cache miss is produced. Branches found during this process are also predicted, thus, not all the filled instructions are from the correct path.

d) Perfect branch prediction.

Assuming linearity among the three components, each contribution to overall branch misprediction penalty can be calculated as follows:

$$\textit{Pipeline-fill penalty} = \frac{IPC_{b)} - IPC_{a)}}{IPC_{d)} - IPC_{a)}}$$

$$\textit{Window-fill penalty} = \frac{IPC_{c)} - IPC_{b)}}{IPC_{d)} - IPC_{a)}}$$

$$\textit{Serialization penalty} = \frac{IPC_{d)} - IPC_{c)}}{IPC_{d)} - IPC_{a)}}$$

We ran the ten benchmarks from the SPECint95 and SPECint2000 suites that exhibit the highest branch misprediction rates. They were compiled with maximum optimizations by a native Compaq compiler for an Alpha processor. A modified version of the *SimpleScalar's v3.0 sim-outorder* cycle-level simulator [3] was used with an instruction window of 64 entries. The branch predictor is an 8 KB *gshare* whose global history register is speculatively updated.

Figure 1 shows the average IPC for each evaluated scenario, whereas Table 1 shows the contributions of each component for the three evaluated pipeline lengths. As expected, the overall loss is greatly influenced by pipeline length (25%, 33% and 39% for pipelines of 6, 10 and 14 stages respectively). Thus, for a pipeline

[1] The total number of cycles in the branch mispredict path is 5, 9 and 13 respectively, i.e. the number of stages from fetch to writeback, in addition to the recovery penalty cycles (*n*=2).

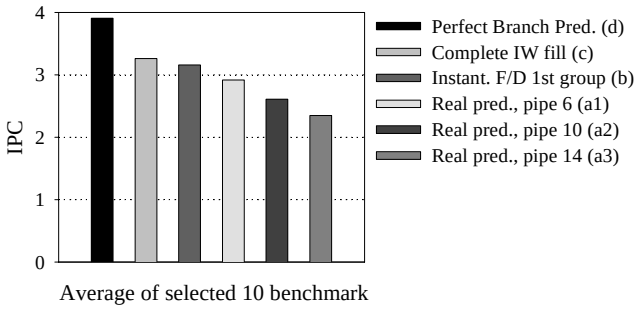

Figure 1. Branch misprediction analysis

Table 1. Average performance loss breakdown for three pipeline lengths of 6, 10 and 14 stages

	Overall loss	Pipeline-fill penalty	Window-fill penalty	Serialization penalty
pipeline 6	25%	25%	10%	65%
pipeline 10	33%	44%	7%	49%
pipeline 14	39%	54%	6%	40%

of 14 stages, the most important component in performance degradation is the *pipeline-fill penalty* (54% of the total overall loss), whereas for a pipeline of 6 stages the most important component is the *serialization penalty* (65% of the total overall loss). Finally, the contribution of the *window-fill penalty* to the overall performance loss, although less substantial, is still significant. For a pipeline of 14 stages the *window-fill penalty* is responsible of 6% of the performance degradation, going up to 10% when the pipeline of 6 stages is considered.

Summarizing, if we consider both *pipeline-fill penalty* and *window-fill penalty*, the aggregate effect is responsible for 35%, 51% and 60% of the overall loss for pipelines of 6, 10 and 14 stages respectively. Hence, by using only multi-path fetching/decoding/renaming, and possibly pre-scheduling (but not execution), approximately half the branch misprediction penalty can be saved. Furthermore, the fraction of the penalty that can be saved increases as pipelines become longer, as is the current trend.

4. DUAL PATH INSTRUCTION PROCESSING

Dual Path Instruction Processing (*DPIP*) is a mechanism for reducing the branch misprediction penalty. Based on the preceding branch misprediction penalty analysis, one of the components that most degrades performance is the *pipeline-fill penalty*. The *DPIP* mechanism fetches, decodes, and renames instructions from the alternative path of some selected low confidence branches. Unlike previous proposals, *DPIP* does not simultaneously execute instructions from both paths. The key point of our proposal is the balance between complexity, cost, and performance benefit. In particular, complexity is reduced when compared with multiple path execution proposals.

Figure 2 depicts a schematic diagram of the *DPIP* hardware. It includes a single fetch unit that can be time-shared by two paths. Thus, at any given cycle, it fetches instructions either from the

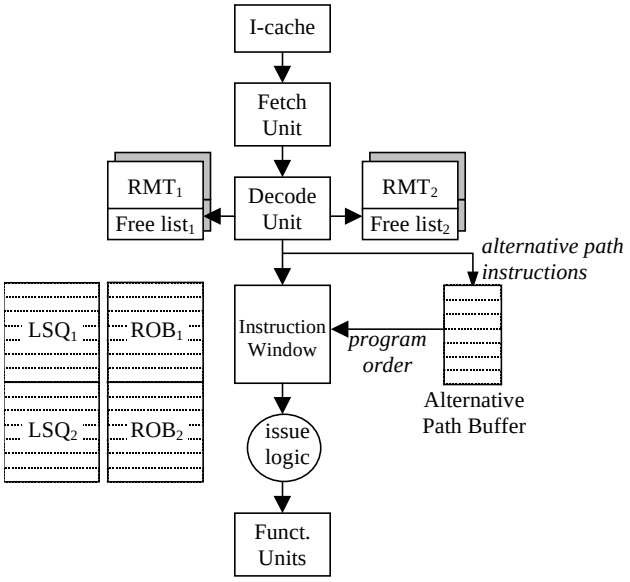

Figure 2. *DPIP* architecture

predicted path or the alternative path. Not all branches are candidates to be forked, and the selection of those that are appropriate is critical for performance. This is accomplished by branch confidence estimation. *DPIP* can only manage two active paths at the same time, and no new branches can be forked until the current forked branch is resolved.

When a branch is forked, a shadow copy of the original register mapping table (RMT_1) is made (this is already done in some processors such as the MIPS R10000). The fetch unit begins working alternatively on both paths, using the appropriate register mapping table: RMT_1 or RMT_2. The reorder buffer (*ROB*) and the load/store queue (*LSQ*) structures are also duplicated. Thus, instructions from the predicted path are inserted and into ROB_1/LSQ_1 and, after decoding, into the main instruction window (*IW*). Instructions from the alternative path are inserted into ROB_2/LSQ_2 and, after decoding, into the *Alternative Path Buffer* (*APBuffer*)[2]. We assume that both *IW* and *APBuffer* have the same number of entries, and half the entries of ROB_1 and ROB_2. The duplication of *ROB* and *LSQ* queues constitutes the major source of hardware cost of our proposal. Note that they are accessed in parallel and independently managed, so it may be assumed that such accesses are not on the critical path. Furthermore, a dual path execution implementation would also have to increase these resources to some extent, and in a way that is on a critical path. In addition, the use of duplicated queues simplifies data movement between ROB_1/LSQ_1 and ROB_2/LSQ_2 after a branch misprediction is detected, while at the same time preserves the program order of the alternative path when using pre-scheduling since the *APBuffer* is not implemented as a queue.

Branches from the alternative path are also predicted using the underlying branch predictor. In order not to interfere, both paths keep a separate copy of the *global history register* of the branch predictor. After a branch misprediction, the wrong path instructions from ROB_1 and LSQ_1 are flushed and RMT_1 is restored with the RMT_2 map. During the penalty cycles dedicated to recovery actions, the remaining entries from ROB_1 and LSQ_1 are inserted into ROB_2 and LSQ_2 respectively. After flushing, ROB_1 and LSQ_1 usually have very few useful entries (experimental results show that, considering a rate of 8 instructions per cycle, up to two cycles will suffice). For the next forked branch, the predicted path will use ROB_2 and LSQ_2, whereas the alternative path will use ROB_1 and LSQ_1. This is accomplished by changing the head/tail pointers of these FIFO structures. When a branch is forked, the fetch unit works on both paths provided that there is room in both queues ROB_1 and ROB_2. If one of them becomes full, the fetch unit continues processing only the predicted path. This ensures that after the misprediction there will be enough space in ROB_2 to accommodate the remaining instructions from ROB_1.

Once the recovery actions have finished, the instructions stored in the *APBuffer* are inserted into the *IW* in program order. As instructions enter the *IW*, they must verify the actual state of their input registers. This is accomplished by checking the *scoreboard* that processors typically use for tracking the availability of register values. The process continues until all the instructions in the *APBuffer* have been introduced in *IW*. The *IW* begins issuing ready instructions to the functional units as soon as the first block of instructions has been received. In order to reduce complexity, the fetch and decode stages are stalled during this transferring process.

4.1 Confidence Estimator

The performance of *DPIP* strongly depends on the confidence estimator's accuracy. If the confidence estimator labels a prediction as low confidence and the prediction turns out to be correct, *DPIP* incurs in a twofold performance degradation. Firstly, the fetch bandwidth is divided between both paths in spite of the fact that instructions belonging to the alternative path are useless. Secondly, a subsequent incoming mispredicted branch will not be forked until the current forked branch has been resolved. To reduce these situations the confidence estimation mechanism should have high SPEC and PVN[3] metrics. This led us to use the confidence estimator proposed for the *Branch Prediction Reversal Unit* (*BPRU*) scheme [1]. This confidence estimator makes use of predicted data values to assess the confidence of branch predictions, using different information from that employed by the branch predictor (usually branch history and branch PC). This method can be very accurate for branches that close loops or pathological *if-then-else* structures. Simulations for a *gshare* branch predictor of 8 KB along with the confidence estimator of 8 KB obtains an average SPEC = 42% and a PVN = 69% for the SPECint2000.

[2] Instructions from the alternative path can be pre-scheduled when entering the *APBuffer* which provides additional performance benefit at the cost of some additional hardware complexity, as section 4.2 shows.

[3] These metrics were introduced by Grunwald *et al* [11] in order to compare the effectiveness of different confidence estimators. SPEC is defined as the fraction of incorrect predictions labeled as low-confidence, whereas PVN is defined as the fraction of low-confidence branches that are finally mispredicted.

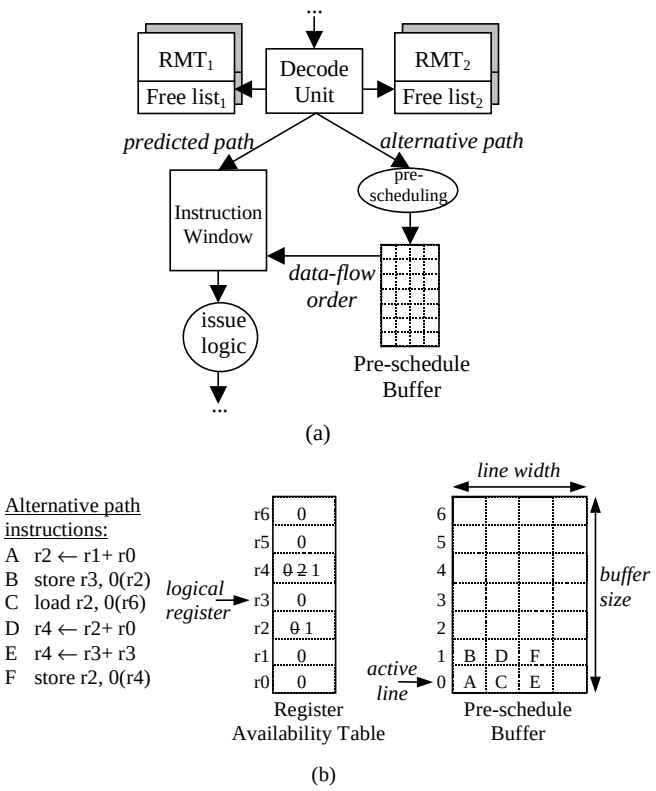

Figure 3. (a) Integrating Pre-Scheduling along with *DPIP* (b) Pre-Scheduling example

4.2 Pre-scheduling Instructions from the Alternative Path

As shown in section 3, the *window-fill penalty* is responsible for a small part of the overall performance degradation after a branch misprediction (10% for a pipeline of 6 stages). As a way of mitigating this penalty, instructions from the alternative path can be *pre-scheduled* in a similar but simplified manner as in [4][23]. With pre-scheduling, correct-path instructions following a misprediction are fed into the instruction window in data-flow order; that is, the first instructions are free of data dependences. This leads to a very high issue rate immediately, even as the window is filling.

Figure 3-(a) shows the *DPIP* architecture extended with the pre-scheduling mechanism. The pre-schedule buffer is implemented as an array of *schedule lines*, each associated with an estimated issue cycle. Each instruction is inserted in the pre-schedule buffer trying to accommodate it in the most convenient slot. The pre-schedule buffer is implemented as a direct-mapped two-dimensional array of *schedule lines*, each one associated with an issue cycle. The schedule line size is equal to the pipeline issue width. For every instruction, its schedule line is determined by:

```
schedule_line = max({reg_availability(input reg1),
                     reg_availability(input reg2)})
reg_availability(output register) = schedule_line +
                                    execution_latency
```

The *register availability* is stored in the *Register Availability Table* (*RAT*), which is indexed by the logical register identifier. The example in Figure 3-(b) shows how instructions are re-ordered depending on the availability of their input registers. All entries in the *RAT* are initialized to 0. Note, for instance, that instruction *B* is scheduled to the *active line* 1 since its input is produced by the previous instruction. Instructions *C* and *E* are independent of all previous instructions from the alternative path, thus they can be scheduled at the first *active line*.

The transfer of instructions from the pre-schedule buffer to the *IW* should be as quick as possible. Since just a single line is moved per cycle, it is good to fill the pre-schedule buffer entries in a compact way, trying to avoid free issue slots in a schedule line as well as empty lines. This can be accomplished by simplifying the proposals in [4][23], by assuming that:

a) All the instructions have an execution latency of 1 cycle (independent of their actual latency), saving bubbles in the pre-schedule buffer.

b) All the instructions before the forked branch have been executed, that is, it is assumed that all *live-in* registers are ready. Initializing all RAT entries to 0 does this.

c) Load and store instructions are pre-scheduled when the input register used to calculate their effective address is ready. The *LSQ* keeps all memory instructions in sequential order, and it enforces memory dependences.

4.3 Optimizations

4.3.1 Delayed forked branches

As stated in [13], branch mispredictions tend to be clustered. Branch misprediction clustering may limit the performance obtained by the *DPIP*, because it is likely to find a new low confidence branch when the alternative path is already active. This will prevent the second branch from forking a new path. One strategy to mitigate this effect is the use of *delayed forked branches*, as in [13]. The idea is to save the processor state (*RMT/Free-list* and PC) when the second low confidence branch is encountered. After the resolution of the current forked branch, when the pre-schedule buffer is empty, the *delayed* branch can be forked using the previously saved state. To simplify the mechanism, only the first low confidence branch fetched by either the predicted or the alternative path can be delayed. For this purpose, *DPIP* uses two *RMT/Free-list* structures for the main path and other two for the alternative path, as Figure 2 shows.

4.3.2 Relaxing Confidence Estimation

DPIP performance can also be improved by tuning the confidence estimator for the dual path application. Preliminary studies showed us that the alternative path was active only 32% of the total execution time. We can augment this time by increasing the low confidence set (labeling more branches as non-confident, i.e. a higher SPEC), at the expense of forking more branches that actually are predicted correctly (a lower PVN). This is accomplished by:

a) Changing the management of the saturating counters of the Confidence Estimator. These counters assign confidence to each branch prediction and are incremented by one if a

branch is mispredicted, and decreased by one otherwise. For every predicted branch, if the corresponding counter of the confidence estimator is above a threshold, the branch fork is initiated. In order to fork more branches, the counters are increased by two and decreased by one.

b) In order to fork a branch, the saturating counters of the branch predictor are also considered. Whenever the branch predictor determines that a branch is either weakly taken or weakly not-taken, a fork action is initiated.

These changes increment the average SPEC to 68% whereas the PVN is reduced to 43% for the SPECint2000.

4.3.3 Unbalanced Fetch Distribution

Forking branches that have been correctly predicted may cause performance degradation due to the limited fetch bandwidth available to the correct path. The selection of a good fetch policy can be tricky and can hide the effect of useless fork actions. Therefore, we choose to categorize the confidence of branches in three ways rather than using the conventional two (high/low).

a) Strong non-confident predictions: there is a high likelihood of a misprediction. Thus, both paths should be followed, and the fetch should be balanced with alternating cycles of full bandwidth for each path. The confidence estimator indicates this situation when the counter is above the threshold.

b) Weak non-confident predictions: for these branches we apply an unbalanced fetch distribution by allocating two cycles to the predicted path and one cycle to the alternative path. Branches predicted as *weakly taken* or *weakly not-taken* belong to this category, independent of the confidence estimator.

c) Confident predictions: these branches are likely to be correctly predicted. Only the predicted path is followed at the full fetch bandwidth.

5. EXPERIMENTAL RESULTS

5.1 Simulation Methodology

To evaluate the performance of *DPIP*, we used the ten benchmarks from the SPECint95 and SPECint2000 suites that exhibit the highest branch misprediction rates. All benchmarks were compiled with maximum optimizations (`–O4 –fast`) by the Compaq Alpha compiler and were run using a modified version of the *sim-outorder* cycle-level simulator included in the *SimpleScalar/Alpha* v3.0 tool set [3]. Due to the large number of dynamic instructions in some benchmarks, we reduced the input data set while keeping a complete execution. Table 2 shows the characteristics of each particular benchmark.

5.2 *DPIP* Performance

This section presents an evaluation of the proposed *DPIP* mechanism in an 8-way superscalar processor. Table 3 shows the configuration of the simulated architecture. The modeled pipeline has been lengthened to 14 stages (from fetch to commit), following the pipeline scheme of the IBM Power4 processor [19]. Section 5.3.4 evaluates the influence of the pipeline depth on the performance of the *DPIP* architecture.

Table 2. Benchmark characteristics

	benchmarks	input set	simulated instruc. (Mill.)	dyn.cond. branches (Mill.)	gshare 16 KB miss-rate
Spec95	compress	40000 e 2231	170	13	9.9%
	gcc	genrecog.i	145	19	8.0%
	go	9 9	146	15	16.7%
	ijpeg	specmun -qual 45	166	9	8.1%
Spec2000	bzip2	input.source 1	500	43	7.8%
	crafty	test (modified)	437	38	6.6%
	gzip	input.source 1	500	52	8.7%
	mcf	test	259	31	7.4%
	twolf	test	258	21	9.7%
	vpr	test	500	45	7.1%

Table 3. Configuration of the simulated processor

Fetch engine	Up to 8 instr/cycle, 2 taken branches, 2 cycles of misprediction penalty.
Issue engine	Issues up to 8 instr/cycle, 128-entries reorder buffer, 64-entries load/store queue.
Pre-scheduling	Up to 64 instructions, 8 instr/schedule-line
Functional Units	8 integer alu, 2 integer mult, 2 memports, 8 FP alu, 1 FP mult.
L1 Instr-cache	64 KB, 2-way, 32 bytes/line, 1 cycle hit lat.
L1 Data-cache	64 KB, 2-way, 32 bytes/line, 1 cycle hit lat.
L2 unified cache	512 KB, 4-way, 32 bytes/line, 6 cycles hit latency, 18 cycles miss latency.
Memory	8 bytes/line, virtual memory 4 KB pages.
TLB	128 entries, fully associative.

Figure 4 shows the IPC obtained by *DPIP* for the selected benchmarks as well as the harmonic mean. The underlying branch predictor is a *gshare* [22] whose global history register is speculatively updated. The different experiments performed are:

a) *gshare(single-path)*, which is the baseline single-path microarchitecture without forked branches;

b) *gshare+DPIP*, our proposed architecture model using pre-fetching/decoding/renaming for alternative path instructions;

c) *gshare+DPIP+preSched*, which also pre-schedules alternative path instructions;

d) *gshare+BPRU+DPIP*, which incorporates the *BPRU* engine [1] in conjunction with the *DPIP* architecture[4];

[4] Branch prediction reversal has been demonstrated as a simple approach to improving prediction accuracy [1][21]. Such proposals reverse the predicted outcome of the low confidence predictions, whereas *DPIP* is based on processing those alternative paths predicted with low confidence. Therefore, both mechanisms can be complementary by reversing every branch predicted with low confidence at the same time they are also forked.

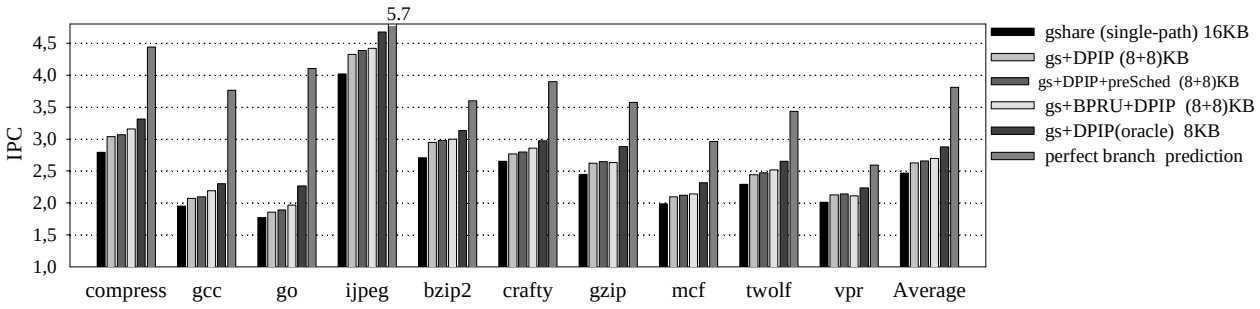

Figure 4. *DPIP* performance using a *gshare* branch predictor

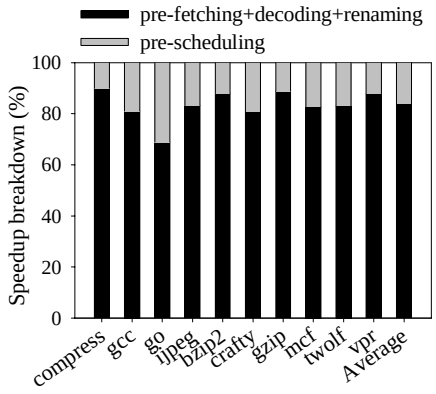

Figure 5. *DPIP* speedup over single-path breakdown

e) *gshare+DPIP(oracle)*, assuming a perfect confidence estimator, that is, only the mispredicted branches are labeled as low confidence; and finally,

f) Perfect branch prediction.

In all experiments we assume equal total table size of 16 KB, including the confidence estimator. In this case, the *gshare* and the confidence estimator are 8 KB each.

The overall loss due to branch mispredictions (the difference between the single-path and the perfect branch prediction experiments) is 35% on average. The *DPIP* architecture using just pre-fetching/decoding/renaming, achieves improvements on all benchmarks over the single-path scheme. On average, the obtained speedup for the ten selected benchmarks is 7% (up to 9% for *bzip2*).

When the pre-scheduling mechanism is also included in *DPIP*, additional improvements are obtained. On average, the obtained speedup for the ten selected benchmarks is 8% (up to 10% for *bzip2*). Figure 5 breaksdown speedups considering both alternatives: a) speedup obtained by pre-fetching/ decoding/renaming, and b) speedup obtained by pre-scheduling. On average, pre-fetching/decoding/renaming is responsible of 84% of the total speedup, while the benefit provided by pre-scheduling represents 16% of the total speedup. For the *go* benchmark, pre-scheduling accounts for 31% of the total speedup. Therefore, the major performance benefits are provided by performing only pre-fetching/decoding/renaming of alternative path instructions. However, the use of pre-scheduling still provides additional benefits at the cost of a relatively small increment in hardware complexity, as previously showed in section 4.2.

We can also observe in Figure 4 that branch prediction reversal together with forked branches provides important benefits: *gshare+BPRU+DPIP* obtains an average IPC improvement of 10% over the single-path scheme (up to 13% for *compress*), which shows that both schemes can be complementary.

Finally, the performance of *DPIP* depends on the confidence estimator's accuracy. For this purpose, the experiment labeled as *gshare+DPIP(oracle)*, uses an oracle confidence estimator in order to provide a measure of the potential of *DPIP*. We can observe in Figure 4 that a perfect confidence estimator provides substantial improvements for all benchmarks: an average speedup of 17% (up to 28% for *go*). This shows the potential of the *DPIP* architecture, and suggests that there is still an opportunity for improvement by using better confidence estimators or improving the fetch policies.

5.3 Sensitivity Study

This section studies the performance of the *DPIP* mechanism when some architectural parameters of the processor are varied, and other parameters are maintained as in previous section. We present average performance results for the single-path scheme, the proposed *DPIP* architecture with pre-scheduling, and finally, *DPIP* with an oracle confidence estimator.

5.3.1 Branch Predictor and Confidence Estimator Size

The first group of experiments concerns the size of the *gshare* branch predictor and the size of the confidence estimator. The studied total size ranges from 8 KB to 64 KB. As in the previous section, the *gshare+DPIP* experiment devotes half of the total size to the branch predictor and the other half to the confidence estimator. The *gshare+DPIP(oracle)* experiment only uses a *gshare* whose size is half of the total size.

The single-path model obtains an average misprediction rate that ranges from 9.8% (8 KB) to 7.9% (64 KB). Figure 6 shows that the speedup of *DPIP* over single-path slightly improves as size grows, obtaining speedups from 7.8% (8 KB) to 8.5% (64 KB). When the oracle confidence estimator is considered, these speedups increase to 17.6% and 15.7% respectively. If we consider iso-performance lines, a *gshare+DPIP* of 8 KB

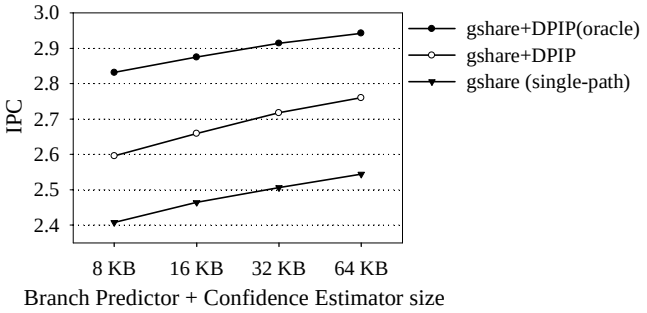

Figure 6. Average IPC for different total sizes of the branch predictor and confidence estimator

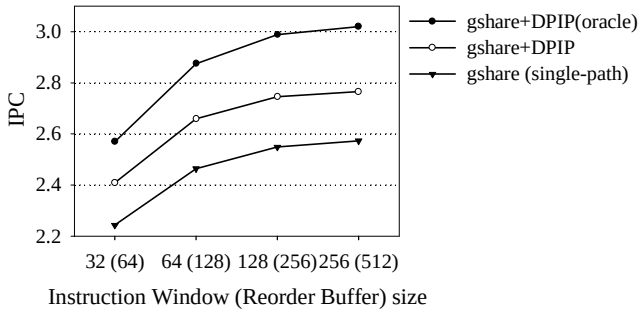

Figure 8. Average IPC for different instruction window sizes. Reorder buffer sizes are in brackets

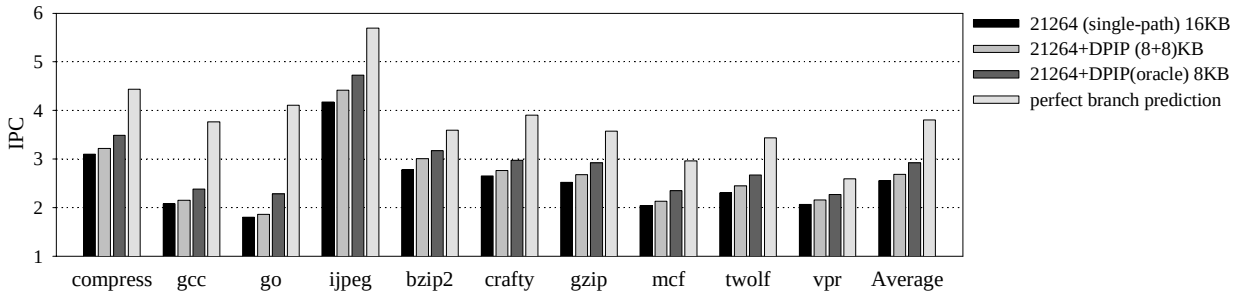

Figure 7. Evaluation of the *DPIP* performance using the Alpha 21264 branch predictor

outperforms the single-path model of 64 KB (IPCs of 2.6 and 2.54 respectively), therefore single-path needs table sizes 8 times larger to achieve about the same performance.

5.3.2 Using DPIP *with Other Underlying Branch Predictor: Alpha 21264 branch predictor*

This experiment shows the performance benefits that the *DPIP* mechanism provides when a more sophisticated branch predictor is used. We use the branch predictor of the Alpha 21264 processor [16] due to its high accuracy[5]. As in section 5.2, we always compare configurations with equal total table sizes of 16 KB, including the confidence estimator size.

First, the average misprediction rate is 8.1%, which is better than the obtained in section 5.2 using a *gshare* of 16 KB (9.0%). Thus, the new average overall loss due to branch mispredictions is reduced to 33%. In Figure 7 we can see that, again, the *DPIP* architecture achieves improvements on all benchmarks over the single-path scheme. On average, the obtained speedup is 5%. The highest speedup is 8% for *bzip2*, whereas the lowest improvement is 4% for *go*, since this benchmark is the one which experiences the highest reduction in misprediction rate when the 21264 branch predictor is used. Finally, when an oracle confidence estimator is considered, the average speedup increases to 15% (up to 27% for *go*), which shows the great potential of *DPIP* even when better branch prediction schemes are used.

[5] The Alpha 21264 branch predictor is composed by a metapredictor that chooses between the predictions made by a global GAg predictor and a local PAg predictor.

5.3.3 Instruction Window Size

This group of experiments evaluates the effect of the instruction window size on the performance of the *DPIP* architecture. The studied window sizes ranges from 32 to 256 entries. In all cases we have considered a reorder buffer whose number of entries doubles that of the instruction window. Figure 8 shows that the speedup of *DPIP* over single-path is almost the same as window size grows, obtaining speedups from 7.3% (32 *IW*-entries) to 7.5% (256 *IW*-entries). In the case of *gshare+DPIP(oracle)*, this speedups increase to 14.5% and 17.5% respectively. It also can be observed that performance benefit almost saturates with an *IW* of 256 entries.

5.3.4 Pipeline Depth

Finally, we evaluated the effect of pipeline depth on the performance of the *DPIP* architecture model. As stated in the branch prediction penalty analysis of section 3, *pipeline-fill penalty* is one of component that most degrades performance. We varied the pipeline depth by changing only the number of stages of the in-order front-end (fetch/decode). Figure 9 shows the average IPC for total pipeline lengths (from fetch to commit) of 10, 14, 16 and 20 stages. We can see that the *DPIP* improvement over single-path increases as the pipeline depth grows. Thus, for 10-stage pipeline, *gshare+DPIP* obtains an average speedup of 4.1% over the single-path model, whereas for 20 stages (like the Pentium 4) the obtained speedup is 12.2%. In the case of *gshare+DPIP(oracle)*, these speedups are 9.2% and 25.3%, respectively. Note that, in a superscalar processor after a misprediction, instructions from the correct path must be fetched

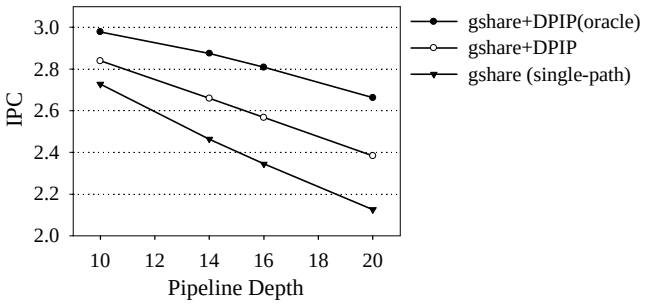

Figure 9. Average IPC for different pipeline depths

and decoded, which takes several cycles. With our proposal, these instructions have been already fetched, decoded and renamed which effectively hides the *pipeline-fill penalty*.

6. CONCLUSIONS

One of the major sources of the performance degradation due to branch mispredictions is the *pipeline-fill penalty*. We propose *Dual Path Instruction Processing* (*DPIP*), a mechanism that alleviates the penalties introduced by branch mispredictions by attempting to maintain a high execution throughput immediately after a misprediction. We propose fetching, decoding and renaming instructions from the alternative path for low confidence predicted branches, at the same time as the predicted path is being executed. In addition, the *window-fill penalty*, can be also reduced by *pre-scheduling* alternative path instructions and by placing them in a special buffer in data-flow order. However, the instructions are not issued for speculative execution.

The key point of our proposal is the balance between complexity, cost, and performance. Complexity is significantly reduced with respect to multiple path execution proposals. The performance obtained by both approaches is well below that provided by perfect branch prediction. However, previous work [18] reported an average speedup of 10.4% over a single-path organization for a pipeline of 10 stages with a dual-ported cache and duplicating the fetch and decode bandwidth. Using a single-ported cache the speedup is 0.6%, mainly due to the aggressive confidence estimator, which was not specially tuned, as mentioned by the authors. Our proposal obtains a speedup of 4.1% for a pipeline of 10 stages using a single-ported cache, due not only to the confidence estimator but to the use of delayed forked branches. In addition, resources such as the register file, the data cache or the functional units are not wasted on instructions that will be later discarded, as happens with the multiple path execution mechanisms.

The *DPIP* with a *gshare* of 8 KB and a confidence estimator of 8 KB, shows an average performance improvement of 8% for a 14-stage superscalar processor over a single-path scheme with a *gshare* of 16 KB (up to 10% for some applications). The major performance benefits are provided by just pre-fetching/decoding/renaming alternative path instructions: 84% of the total speedup. On the other hand, pre-scheduling accounts for 16% of the total speedup, providing additional benefits at the cost of a relatively small increment in hardware complexity.

Finally, the performance of the *DPIP* is affected by the accuracy of the confidence estimator: with oracle confidence estimation the average speedup is 17% (up to 28% for some applications). We have also shown that the performance of the *DPIP* is robust against modifications of some architectural features, being pipeline depth the one that influences the most in its performance.

7. ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Science and Technology under grant TIC2000-1151-C07-03, by the Spanish CICYT under grant TIC98-0511, by the National Science Foundation grant CCR-9900610, by Intel Corporation and IBM Corporation.

8. REFERENCES

[1] J.L. Aragon, J. Gonzalez, J.M. Garcia and A. Gonzalez. “Confidence Estimation for Branch Prediction Reversal”. *Proc. of the Int. Conference on High Performance Computing*, pp. 214-223, Dec. 2001.

[2] J. Bondi, A. Nanda and S. Dutta. “Integrating a Misprediction Recovery Cache into a Superscalar Pipeline”. *Proc. of the Int. Symp. on Microarchitecture*, 1996.

[3] D. Burger and T.M. Austin. “The SimpleScalar Tool Set, Version 2.0”. Technical Report #1342, University of Wisconsing-Madison, Computer Sciences Department, 1997.

[4] R. Canal and A. Gonzalez. “A Low-Complexity Issue Logic”. *Proc. of the Int. Conference on Supercomputing*, 2000.

[5] P.Y. Chang, M. Evers and Y.N. Patt. “Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference”. *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, 1996.

[6] W.D. Connors, J. Florkowski and S.K. Patton. “The IBM 3033: An Inside Look”. *Datamation*, pages 198-218, May 1979.

[7] J. Cortadella and J.M. Llabería. “An Intelligent IFU for Pipelined Processors that Make Control Instructions Transparent to the Execution Unit”. *Proc. of Int. Symp. on Applied Informatics*, pp. 188-191, Feb. 1987.

[8] P.N. Glaskowsky. “Pentium 4 (Partially) Previewed”. *Microprocessor Report*, August 2000.

[9] A. González, J.M. Llabería and J. Cortadella. “A Mechanism for Reducing the Cost of Branches in RISC Architectures”. *Microprocessing and Microprogramming*, vol. 24,1-5, pp. 565-572, Aug. 1988.

[10] J. González and A. González. “Control-Flow Speculation through Value Prediction for Superscalar Processors”. *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, 1999.

[11] D. Grunwald, A. Klauser, S. Manne and A. Pleszkun. “Confidence Estimation for Speculation Control”. *Proc. of the Int. Symp. on Computer Architecture*, 1998.

[12] L. Gwennap. “MIPS R10000 Uses Decoupled Architecture”. *Microprocessor Report*, pp.18-22, Oct. 1994.

[13] T.H. Heil and J.E. Smith. “Selective Dual Path Execution”. Technical Report, University of Wisconsin-Madison, ECE, 1997.

[14] E. Jacobsen, E. Rotenberg and J.E. Smith. “Assigning Confidence to Conditional Branch Predictions”. *Proc. of the Int. Symp. on Microarchitecture*, 1996.

[15] D.A. Jiménez, S.W. Keckler and C. Lin. “The Impact of Delay on the Design of Branch Predictors”. *Proc. of the Int. Symp. on Microarchitecture*, 2000.

[16] R.E. Kessler, E.J. McLellan and D.A. Webb. “The Alpha 21264 Microprocessor Architecture”. *Proc. of the Int. Conf. on Computer Desing*, 1998.

[17] A. Klauser, A. Paithankar and D. Grunwald. “Selective Eager Execution on the PolyPath Architecture”. *Proc. of the Int. Symp. on Computer Architecture*, 1998.

[18] A. Klauser and D. Grunwald. “Instruction Fetch Mechanisms for Multipath Execution Processors”. *Proc. of the Int. Symp. on Microarchitecture*, 1999.

[19] K. Krewell. “IBM’s Power4 Unveiling Continues”. *Microprocessor Report*, November 2000.

[20] C.C. Lee, I.C.K. Chen and T.N. Mudge. “The Bi-Mode Branch Predictor”. *Proc. of the Int. Symp. on Microarchitecture*, 1996.

[21] S. Manne, A. Klauser and D. Grunwald. “Branch Prediction using Selective Branch Inversion”. *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, 1999.

[22] S. McFarling. “Combining Branch Predictors”. Tech. Report #TN-36. Digital Western Research Lab., 1993.

[23] P. Michaud and A. Seznec. “Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”. *Proc. of the High-Perf. Computer Architecture*, 2001.

[24] J.E. Smith. “A Study of Branch Prediction Strategies”. *Proc. of the Int. Symp. on Computer Architecture*, pp. 135-148, 1981.

[25] D.M. Tullsen, S.J. Eggers and H.M. Levy. “Simultaneous Multithreading: Maximizing On-chip Parallelism”. *Proc. of the Int. Symp. on Computer Architecture*, 1995.

[26] A.K. Uht and V. Sindagi. “Disjoint Eager Execution: An optimal Form of Speculative Execution”. *Proc. of the Int. Symp. on Microarchitecture*, 1995.

[27] S. Wallace, B. Calder and D.M. Tullsen. “Threaded Multiple Path Execution”. *Proc. of the Int. Symp. on Computer Architecture*, 1998.

[28] T.Y. Yeh and Y.N. Patt. “Two-Level Adaptive Branch Prediction”. *Proc. of the Int. Symp. on Microarchitecture*, pp. 51-61, 1991.