DCE: The Dynamic Conditional Exectution in a Multipath Control Independent Architecture

Rafael dos Santos
Philippe Navaux
Mario Nemirovsky

UCSC-CRL-01-08
June 2001

Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

On-chip large tables allow to record history of past branches to later use it to predict the outcome of future branches. Even though branch predictors reach a good performance, misprediction penalty and cache alignment problems still represent a limit for performance.

Our analysis brings a new insight into branches. We show that some techniques can be applied in order to take advantage of locality. We then present an analysis of 18 benchmarks of the SPEC95 suite and we show that techniques like multipath execution can be implemented with no significant impact in the instruction cache as previous works were concerned about.

We also present a concept called DCE - *Dynamic Conditional Execution*. DCE allows to fetch multiples paths within one single cache access, single pass, when certain conditional branches are found. These paths can be conditionally executed using multipath techniques, reducing misprediction penalties. Because DCE takes advantage of locality features, cache misalignment effects can be reduced as well. Furthermore, control independence may be exploited in order to avoid to squash useful work when mispredictions are detected. Thus, DCE improves the benefit of multipath execution and speculation as a form to reduce penalties associated with control flow dependencies.

Keywords: Superscalar, Branch Prediction, Multipath, Conditional Execution, Dynamic Predication

Contents

1. Introduction

Constant advances in semiconductor technology have allowed computer architects to slightly increase the complexity of current microprocessor's implementations. Indeed, upcoming advances will allow even more complexity to be added since a larger number of transistor is still possible. According with [NAT97] it is expected to have 1.4 billion transistor available running at 10 GHz by 2012.

Even tough semiconductor technology still promises significant integration, its advance towards the limits imposed by physics has became more important. An important issue is what to do with such a large amount of transistors and how to do it efficiently.

ILP *Instruction Level Parallelism* is nowadays employed in all microprocessors under sort of different techniques. Recent architectures are extracting and exploiting the ILP from single and multiple threads of control.

Better performance has been successfully obtained from advances in the implementation techniques which allows circuits to execute tasks faster and from advances in the architectures and algorithms which has been allowing more parallelism, i.e., more tasks to be executed at the same time.

The combination of these two key things result in more tasks being executed simultaneously within smaller fractions of time (faster). Although we have seen remarkable improvements in the IC design process, we can not foresee exactly what kind of architectures will be implemented over those transistors in the next few years.

So many ideas have been proposed recently from executing multiple paths of conditional branches to executing multiple threads and processes in a single one-chip processor. All of them rely on fetching more instructions to feed starving functional units.

Fetching more instructions is complicated because correct instruction addresses are completely unknown during fetch after a conditional branch. Branch prediction is applied to predict those addresses thus avoiding to stall the fetch unit on the presence of conditional branches.

Two-level branch prediction [YEH91] can predict correctly about 97% of time for certain benchmarks. Other schemes suggest that combining different branch predictors can help in the case of a hard-to-predict branch is fetched. Hybrid schemes perform well than single schemes but imply also higher costs.

Branch prediction is currently playing an important role in the fetch process. Without branch prediction would be impossible to fetch instructions in the demand required by current microprocessors. Despite the correctness of branch predictors, the remaining small misprediction rates degrade considerably the performance.

Another limit is also imposed by alignment problems because branch targets are rarely positioned at the beginning of a cache line. After redirect the fetch to the target of a branch, in case the branch is taken (or predicted as taken), bandwidth is often wasted because instructions that are physically in the line but before the target are discarded.

In the case the branch is not-taken, the same line is sometimes read twice from the instruction cache to fetch the instructions along the not-taken path which are sequential to the branch itself. In both cases, bandwidth is wasted either by discarding the instructions that come within the cache line but before the target or by fetching twice the same cache line.

Several problems affect the performance of the fetch engine which reflects negatively into the final overall performance. This work is concentrated on the study of the problems limiting the fetch of wide deep superscalar pipelines.

Some conclusions are highlighted and a proposal called DCE which stands for Dynamic Conditional Execution is presented. A matter of fact is that even using high performance branch predictors, the small fractions of mispredictions harm the overall performance significantly.

DCE is proposed as an alternative to reduce the occurrence of mispredictions and the penalties associated with the recovery and squash of instructions after conditional branches. DCE targets short branches where most of mispredictions are concentrated.

DCE executes eagerly (i.e. both paths) those branches in order to eliminate mispredictions in these situations. Less branches are predicted at the fetch time but more instructions are issued for execution. Thus the fetch engine tends to be slightly simple than conventional machines but more pressure is moved into the execution core.

Conventional machines commit less instructions than they are able to. This happens because some instructions are executed and threw away completely after a misprediction. DCE employs an hybrid model involving multipath, control independence and selective squash on certain cases. The present work is organized as follows:

Chapter 2 introduces the base machine and the benchmarks characteristics used in the experiments. Chapter 3 presents an initial study and evaluation of the negative effect of mispredictions and the role of branches in wide deep superscalar pipelines.

Chapter 4, summarizes the effects of mispredictions and discusses some of the issues on the treatment of them and solutions proposed previously based on multipath schemes. The chapter is centered on the discussion of Pros and Cons of multipath.

Chapter 5 introduces DCE as a proposal for reducing mispredictions in short branches. DCE is discussed briefly by presenting the basic concepts and the motivations for the proposal. The concepts are presented and some expected results are delineated. Chapter 6, Revisiting Branches, presents a new insight into branch analysis addressing mainly the locality of certain branches, where mispredictions are concentrated, which is the base for this proposal.

Finally, Chapter 7 introduces a more detailed concept about DCE and discusses the microarchitecture involved in its implementation. Each aspect of the architecture is discussed and the future steps are suggested.

2. The Architecture and Benchmarks

2.1 Benchmarks

For the experiments presented in this chapter we simulated all benchmarks of the SPEC95 suite. Both integer and float point benchmarks were simulated up to 200 million instructions but only the latest 100 million instructions were counted on the performance graphs. The first 100 million instructions were skipped.

The table 2.1 shows each benchmark and the percentual of each type of instruction executed. Notice that we divided the range of instructions in 4 types: branches, loads, stores and other instructions. Branches represents both conditional and unconditional branches hereby called jumps.

The frequency of occurrence of each type of instruction was gathered from the execution stage of the pipeline because we intended to capture the actual number of instructions that were in-flight. The numbers presented in the table show the percentage of instructions of each type that were executed but not necessarily committed. Because the architecture used a conventional branch predictor (i.e. not perfect), instructions from the wrong-path of mispredicted branches also consumed resources.

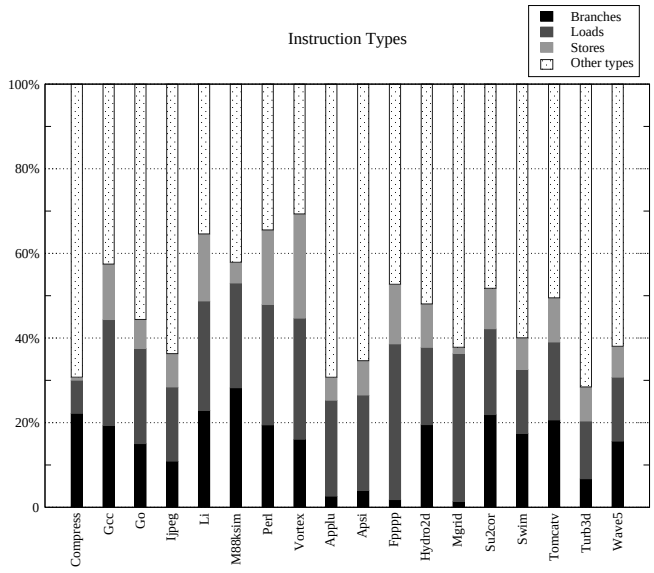

Figure 2.1: Classes of Instructions

We highlighted the five highest indexes among all the benchmarks for branches, loads and stores. For branches, we observed frequencies of 28.15, 22.80, 22.10, 21.81 and 20.55 percent of all executed instructions for the benchmarks *M88ksim*, *Li*, *Compress*, *Su2cor* and *Tomcatv*, respectively.

For loads, the five highest frequencies were 36.80, 34.94, 28.63, 28.46 and 25.87 percent of all instructions executed in the benchmarks *Hydro2d*, *Fpppp*, *Vortex*, *Perl* and *Li*, respectively. As for stores, 24.63, 17.59, 15.82, 14.07 and 13.09 percent of all instructions for the benchmarks *Vortex*, *Perl*, *Li*, *Gcc* and *Fpppp*, also respectively.

Table 2.1: Benchmarks used in the Simulations

Benchmark	Input	Dynamic			
		Branches (%)	Loads (%)	Stores (%)	Other Insn (%)
Compress	bigtest.in	**22.10**	7.83	0.70	69.37
Gcc	-O cp-decl.i -o cp-decl.s	19.22	25.05	**13.09**	42.63
Go	50 21 9stone21.in	14.95	22.53	6.80	55.71
Ijpeg	vigo.ppm	10.80	17.53	7.87	63.80
Li	*.lsp	**22.80**	**25.87**	**15.82**	35.52
M88ksim	ctl.raw	**28.15**	24.75	4.89	42.20
Perl	primes.pl ¡ primes.in	19.38	**28.46**	**17.59**	34.58
Vortex	vortex.raw	15.96	**28.63**	**24.63**	30.78
Applu	applu.in	2.53	22.67	5.42	69.38
Apsi		3.88	22.55	8.11	65.46
Fpppp	natoms.in	1.72	**36.80**	**14.07**	47.41
Hydro2d	hydro2d.in	19.43	18.29	10.23	52.06
Mgrid	mgrid.in	1.32	**34.94**	1.45	62.29
Su2cor	su2cor.in	**21.81**	20.28	9.55	48.36
Swim	swim2.in	17.34	15.15	7.45	60.06
Tomcatv	tomcatv.in	**20.55**	18.41	10.44	50.59
Turb3d	turb3d.in	6.65	13.64	8.04	71.66
Wave5	wave5.in	15.53	15.12	7.29	62.06
Average					
int		19.17	22.58	11.42	46.82
fp		11.07	21.78	8.20	58.93
int & fp		14.66	22.13	9.63	53.55

Figure 2.1 shows the instructions types for each benchmark as a set of four different types: branches, loads, stores and other types. We choose arbitrarily to highlight the five highest indexes, however there is no special reason to choose this number but help to simplify the analysis.

2.2 Reference Architecture

The architecture simulated is a 12 stage pipeline with fetch, virtual1, virtual2, decode/dispatch, issue/execute, virtual3, virtual4, virtual5, virtual6, virtual7, write-back and commit stages.

Virtual stages were included in the pipeline to emulate a deeper pipeline (figure 2.2). Basically the pipeline has only 5 functions which are fetch, decode/dispatch, issue/execute, write-back and commit. We added these extra stages to emulate current designs, where usual pipeline depth ranges from 10 to 15 stages or more.

Below we detailed each aspect of the configuration used in the experiments.

- L1 I-cache: 1 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU replacement policy;
- L1 D-cache: 1 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU replacement policy;
- L2 Unified cache: 6 cycles hit; 512Kb (1024 sets; 4 lines; 128 bytes line); LRU replacement policy;

- Main Memory: 18 cycles first chunk; 1 cycle intermediate chunks; 128 bytes access bus;
- Fetch: 256 entries fetch queue; 16 instructions fetch bandwidth;
- Decode: 16 instructions decode bandwidth;
- Issue: 16 instructions issue bandwidth;
- Commit: 16 instructions commit bandwidth;
- Instruction window size: 256 instructions
- Branch prediction: Hybrid; 2048 entries meta-table; two-level gshare xor; 12 bits history;
- Branch Target Buffer: 512 sets; 4-way;
- Return Address Stack: 32 entries;
- Functional Units:
  - Integer ALU's (ialu): 16 FUs;
  - Integer multiplier/divider (imult): 16 FUs;
  - Float Point ALU's (fpalu): 16 FUs;
  - Float Point multiplier/divider (fpmult): 16 FUs;
- Memory ports: 4 ports;

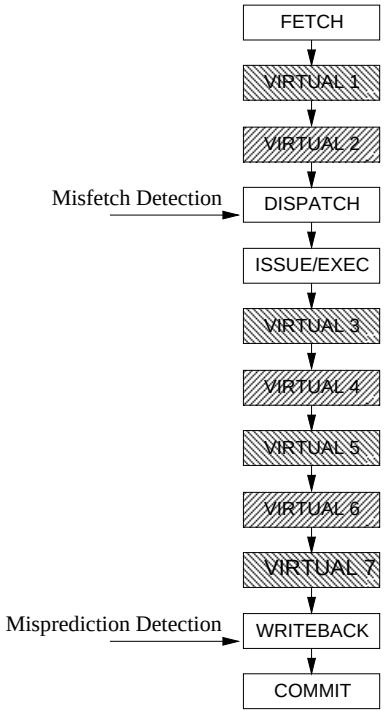

Figure 2.2: Pipeline Structure

The functionality of each stage is described below. Notice that virtual stages are not described because they are essentially delays. If the pipeline is completely full, then virtual stages do not interfere neither with the execution nor the performance. When the pipeline is being filled (e.g. after a misprediction) then those virtual stages count as real delays increasing the time to fill the pipeline.

As we didn't introduce any modification in the simulator besides the virtual stages the following description is similar as presented in [BUR96].

2.2.1 Fetch

The fetch takes the PC address and probes the i-cache to access the respective line. The i-cache has only one port and is blocked until the data is brought from the higher memory levels, in the occurrence of a miss.

Taken branches interrupt the fetch offering a major constraint for the performance. Even if 16 instructions hit in the cache a taken branch will break the sequence of instructions transferred to the dispatch queue.

The predictor is also probed in order to inform the address to access in the next cycle. If the address is unknown the fetch continue fetching through the next sequential address, considering the last instruction transferred to the dispatch queue.

The fetch can be interrupted by the execution unit when a misprediction is detected. In this case, one cycle stall is introduced to recovery the fetch and start to fetch the correct address sent from the execution units. All instructions fetched after the mispredicted branch are squashed before the fetch recovery.

2.2.2 Decode/dispatch

In this stage the instructions fetched are decoded, the register renaming is performed and RUU (Register Update Units) units are allocated to hold each instruction. The dispatch places as many instructions up to the dispatch width in the scheduler queue.

2.2.3 Issue/execute

This stage tracks memory and register dependencies issuing instructions to the execution units when operands are ready and dependencies are satisfied.

Instructions ready are placed in the scheduler queue. Execution units take instructions from the scheduler queue in order to execute them. An instruction retains the functional unit for as many cycles as necessary to execute the instruction according with its latency.

2.2.4 Write-back

In this stage, results produced are propagated to waiting instructions in order to allow them to be issued. Instructions that are waiting for those results are marked as ready to be issued.

Also mispredictions are detected in this stage. In this event, a signal is sent to the fetch indicating a misprediction and the new address to fetch. The pipeline is flushed and a new fetch is started down the correct address.

2.2.5 Commit

This stage does in-order commit, handling instructions that are ready to commit from the write-back stage and updating the d-cache with store values.

The commit keeps retiring instructions until a not ready instruction is at the head of the RUU. When an instruction is committed, its result is placed into the architectural register file and the RUU resources allocated for that instruction are reclaimed.

2.3 Quantifying Misprediction and Misfetch Penalties

A penalty is incurred each time a misfetch or a misprediction occur. In this section we explain how both misfetch and misprediction penalties are charged in our reference architecture and throughout the simulations presented in this work.

A misfetch occurs when a branch is predicted as taken but the target address is not found on the BTB (i.e. BTB miss). That means that the direction of the branch is predicted but the address is unknown. In this case, the fetch can only proceed through the sequential address. This would result in benefit only if the direction was incorrectly predicted, that is the branch is in fact not taken but predicted as taken and the address not found in the BTB.

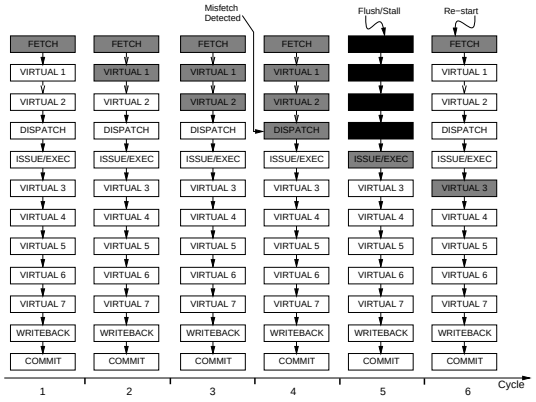

Figure 2.3: Misfetch Penalty

There is also another situation where a misfetch can take place. Sometimes, depending on the scheme used to manage the prediction tables, two or more branches may be mapped to the same entries on the tables. These branches are called conflicting branches because they all access the same entries on the tables. Hence, a branch may use an entry previously accessed by another branch (conflicting) therefore using a wrong address or direction. In our architecture this does not occur. Only BTB misses can generate misfetches.

Moreover, misfetches can only be detected for those branches which carry the target address into the instructions itself, i.e. Direct relative branches. Misfetches resulting from indirect branches or indirect jumps cannot be detected since the target address is usually available only at the execution time.

Figure 2.3 shows a misfetch occurrence. Suppose that on the first cycle, the processor started to fetch from a given address and a branch was found. The predictor was probed and the branch was predicted taken but an address could not be retrieved from the BTB, i.e. BTB miss.

Three cycles later, the branch reaches the dispatch stage and at this time the branch is already decoded. As a BTB miss occurred, the fetch used the sequential address to fetch the next instruction following the branch. The dispatch detects that the branch was predicted as taken but the address used does not match with the target address decoded. At this time the misfetch is detected.

To recover from the misfetch, the processor stalls the fetch at cycle 5 and flushes all instructions that follow the branch inside the pipeline. As all operations are taken in-order until the dispatch, then the branch and all precedent instructions are dispatched and the

rest is discarded. The stages between the dispatch and the fetch are flushed and the fetch is informed of the correct address.

At cycle 6, the fetch starts to fetch again from the correct address. Notice that this assumption is based on the direction predicted. At this time, the branch is not yet resolved and the branch outcome is still unresolved. That means that the direction may be incorrect and perhaps a misprediction can be detected later.

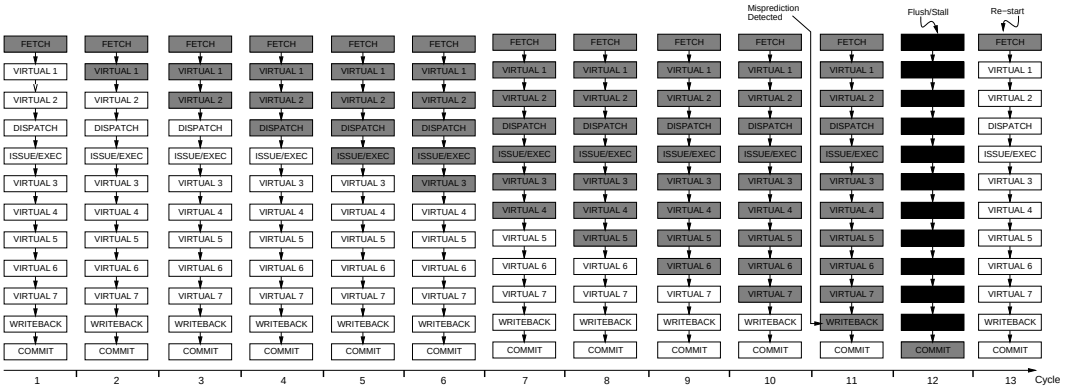

Figure 2.4: Misprediction Penalty

Figure 2.4 shows a misprediction occurrence. Suppose that the processor started to fetch from a given address and found a branch. The branch was predicted taken and an address was retrieved from the BTB. The address passed through the dispatch stage meaning that the correct address was retrieved from the BTB, i.e. there was no misfetch.

However, at cycle 11, the branch is resolved and a misprediction is detected. The recovery procedure is similar to the misfetch recovery but more complex. Because the reference architecture can issue and execute instructions out-of-order, it is possible that some instructions fetched after the mispredicted branch have been already executed.

Therefore, the flush on this case must be selective, i.e. instructions that are logically before the branch cannot be flushed as they may still be waiting for operands and have not yet been executed. In fact, the dispatch queue is flushed so instructions not already dispatched are squashed and those instructions already dispatched are kept into the pipeline.

The dispatch queue and the fetch buffer are flushed and the branch is marked as mispredicted. The fetch is then informed of the correct address and a new fetch is started from that address, at cycle 13.

The instructions that come after the branch are invalidated and their results are not committed to the architectural register file.

Figures 2.3 and 2.4 showed the penalties associated with misfetches and mispredictions as defined in the reference architecture used hereafter.

3. The Role of Branches and Branch Prediction

We conduced some experiments to show how branches harm the performance of a superscalar architecture. In the experiments reported here we used the Simplescalar Tool Set 3.0 [BUR96] to simulate a conventional superscalar architecture, described on section 2.2.

Some of the resources were dimensioned to not limit the performance. The architecture simulated is a 16 way superscalar with a 256 entries instruction window, 16 functional units of each type (ialu, imult, fpalu, fpmult) and 4 memory ports.

The reason to perform these simulations was strictly to understand the behaviour of a wide architecture. We did not propose any modification except to increase some of the resources in order to avoid or, at least, to reduce resource conflicts.

3.1 IPC - Instructions per Cycle

The IPC is the ratio between the number of committed instructions and the cycles spent to execute 100 million instructions:

$$IPC = \frac{number\ of\ instructions\ committed}{cycles\ to\ execute\ 100M\ instructions} \tag{3.1}$$

The number of instructions committed depends on how good were the predictions made throughout the simulation. If a perfect predictor is applied than the number of committed instructions must be the same as the executed instructions. If the predictor fails to predict any branch, then the number of executed instructions will be greater than the number of committed instructions because instructions belonging to the wrong path are executed, but not committed.

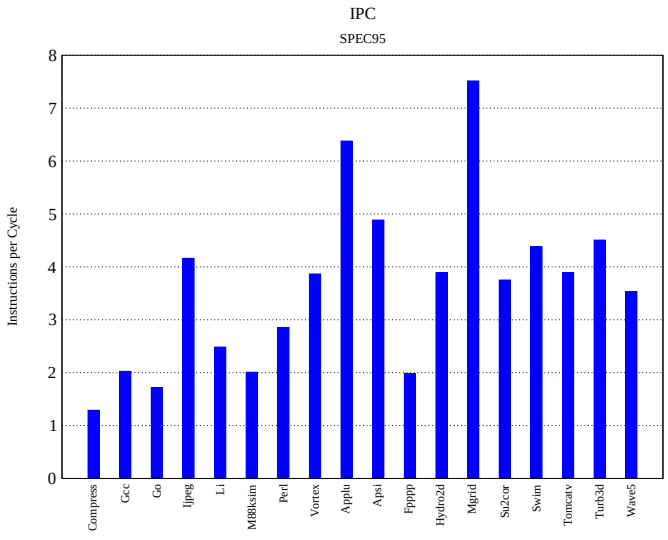

Figure 3.1: IPC of SPEC95

The graph on Figure 3.1 prove that there is a large gap between the performance delivered and the potential performance according to the amount of resources. The architecture

simulated should be able to execute about 16 instructions per cycle. However, the highest IPC was 7.51 instructions per cycle for benchmark *Mgrid* followed by 6.37 instructions per cycle for *Applu*. The highest IPC among the integer benchmarks was 4.16 instructions per cycle for *Ijpeg*.

Thus, we see less than 50% of total expected capacity being used in the best case. The average IPC is only 3.61 instructions per cycle. This means just 22.56% of the potential IPC of 16 instructions based on the architecture width.

3.2 Fetch Limitations and Misprediction Side Effects

The level of parallelism obtained in the execution of a program relies on the efficiency of the fetch. The execution is feed by the scheduler which captures independent instructions and send them to the FUs (Functional Units). The scheduler searches for independent instructions in the instruction window, thus the larger the instruction window the higher the chance to find independent instructions.

When an instruction finishes its execution, the result produced is propagated and the data dependency chain is updated. New instructions become ready to be issued and the scheduler performs again a search for independent instructions. As more functional units become available, it is necessary to increase the size of the instruction window to allow the scheduler to have more options when looking for independent instructions. An efficient instruction window provides useful instructions to be scheduled. However, the fetch must be able to fill the instruction window independent of its size. Obviously, the larger the instruction window the most stressed is the fetch.

Two kind of events can affect the effectiveness of the fetch. Some events stall the fetch by preventing it to perform the accesses to the instruction cache. A miss on the instruction cache, for example, blocks the fetch until the data is brought from the upper levels of the memory system. Of course, a non-blocking cache may allow the fetch to continue in the occurrence of a miss, however the fetch may proceed through an alternative address.

A full fetch buffer also causes the fetch unit to stall. The fetch cannot operate until there is space available to put the instructions brought from the i-cache. We consider cache misses and fetch buffer full as events which prevent the fetch to perform causing stalls.

Other kind of events generate side effects beside the stalls. Mispredictions and misfetch causes the fetch to proceed through wrong addresses generating pollution into the pipeline.

After a branch, the fetch proceed speculatively through a predicted address, because the outcome of such branch instruction is not known until the condition is executed. After to predict the target of a branch, the activity performed by the fetch is conditional. It depends on the outcome of that branch. If the outcome is equal to the prediction made, then the instruction fetched after the branch are useful. If the outcome differs from the prediction then, the instructions fetched after the branch are useless and the eventual results already produced must be thrown away.

In such case, other stages of the pipeline are affected as well because incorrect information is dispersed all over. This is a side effect because not only the time spent to execute useless instructions is lost, but also the work performed is incorrect, which implies a more complex routine to recover to a consistent (i.e. correct) state.

Events such as misprediction and misfetch cause the fetch to stall in order to update to the correct address but also require some sort of recovery to avoid results produced by useless instructions to affect the execution of correct instructions.

3.3 Fetch Performance

3.3.1 Fetch Stalls

I-cache misses and fetch buffer full prevent the fetch to work. Mispredictions and misfetches stall the fetch and invalidate the work already performed affecting other stages of the pipeline as well.

I-cache misses stall the fetch for as many cycles as necessary to bring the data from the higher levels of the memory system. The latency of a miss depends on the location of the data.

A fetch buffer full prevents the fetch to transfer the data from the instruction cache into the pipeline. In this case, the latency will depend on the subsequent stages of the pipeline because they need to consume the instructions that are obstructing the fetch.

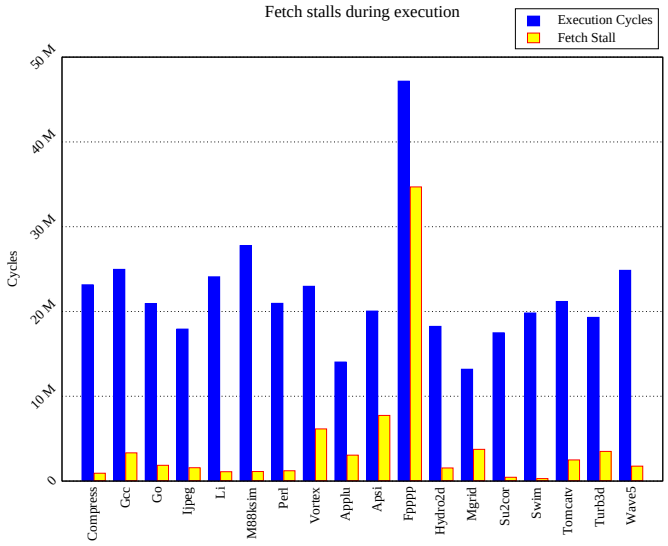

Figure 3.2: Fetch Stalls

Mispredictions and misfetch also cause the fetch to stall. A one cycle stall is introduced to recovery the fetch to the correct address when the misprediction or misfetch is detected. However, mispredictions and misfetches also cause a side effect which imply a major loss. Not only the time needed to recovery the fetch is lost but also the cycles where the fetch brought instructions from the mispredicted addresses.

The penalty caused by mispredictions and misfetches resides essentially on the work squashed. Figure 3.2 shows the number of cycles which the fetch unit was stalled. The black bar shows the total number of cycles spent to execute each benchmark. The gray bar shows the number of cycles where the fetch was stalled.

We see on *Fpppp* that on 73.56% of total time the fetch unit is stalled. On benchmarks *Apsi*, *Mgrid*, *Vortex* and *Applu* we see 38.62%, 28.39%, 26.78% and 21.79% of cycles spent with the fetch unit stalled, respectively.

Figure 3.3 shows how the stall cycles were spent. For each benchmark we show the percentage of cycles spent with each of the four stall causes. The black bar represents

mispredictions, whereas the dark gray represents misfetches. The light gray represents i-cache misses and the hatching bar represents the fetch buffer full.

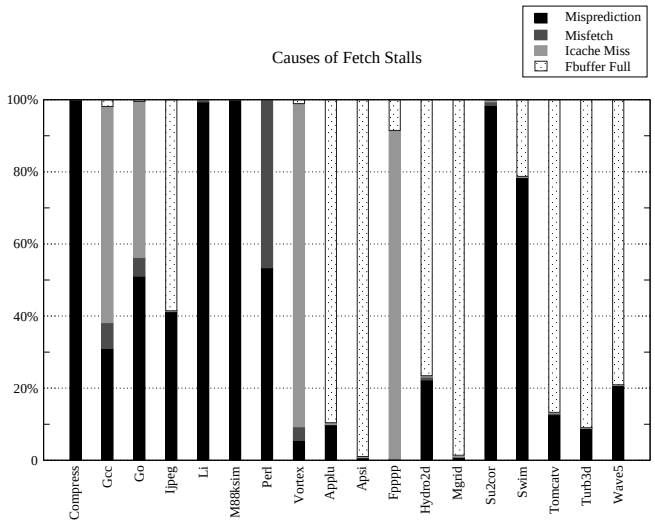

Figure 3.3: Causes of Stalls

For the benchmarks *Fpppp* and *Vortex* the stalls were mainly caused by i-cache misses as for *Gcc* and *Go*. For benchmarks *Compress*, *Li*, *M88ksim*, *Su2cor* and *Swim* the main cause of stall was the occurrence of mispredictions. For these benchmarks the total number of stalls did not go over 8.3% of total execution cycles.

For all other benchmarks simulated the fetch stalled mainly due to the fetch buffer full. That happened for 7 of 10 float point benchmarks (*Applu, Apsi, Mgrid, Tomcatv, Turb3d* and *Wave5*). The fetch buffer caused most of the stalls for only one integer benchmark, *Ijpeg*. In the case of *Perl* benchmark we saw also almost 50% of stalls caused by misfetches.

3.3.2 Effective Loss

In the previous graphs we showed that stalls can limit the performance of the fetch by preventing it to work. The reasons for the fetch to stall were presented and discussed. In this section we show that even few mispredictions can harm the performance of the fetch by causing work performed to be squashed.

In the previous figures we showed the cycles were the fetch was inactive. In Figure 3.4 we are showing all cycles lost due to the same four events: i-cache misses, fetch buffer full, mispredictions and misfetches.

The black bar presents the total number of execution cycles. On the gray bar we have the fetch cycles lost. A cycle lost is either a cycle where the unit was inactive (stalled) or performed an activity which was nullified afterwards.

Any cycle invalidated or inactive is counted as a loss for the fetch unit. We observed that a large portion of the total execution time is spent with the fetch unit either stalled or fetching instructions from the wrong path of mispredicted branches. For the *Swim* benchmark we observed the lowest loss achieving for 24.40% of total execution cycles. For

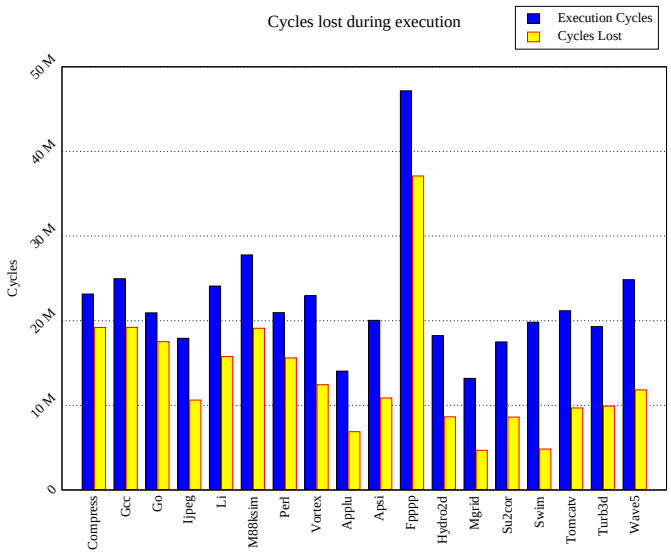

Figure 3.4: Cycles Lost

Go benchmark we observed the worst case where *83.72%* of the total cycles were spent with the fetch unit fetching useless instructions or staying simple blocked. For all benchmarks, except *Swim*, the fetch loss represented more than 40% of total execution cycles.

Table 3.1 summarizes the prediction accuracy and misprediction rate for each benchmark.

Here again we highlighted the five highest indexes for accuracy and for misprediction. We have then benchmarks *Swim*, *Hydro2d*, *Tomcatv*, *Perl* and *Su2cor* presenting the best predictor's performance among all benchmarks. Respectively we have 99.36%, 99.14%, 99.07%, 98.93% and 98.80% as the best indexes reached in this set of experiments. Notice that only *Perl* is an integer benchmark.

On the other hand, we have *Go*, *Applu*, *Compress*, *Ijpeg* and *Gcc* as the benchmarks which presented the highest misprediction rates with indexes of 15.89%, 15.27%, 15.06%, 10.03% and 7.94%, respectively.

The figures on this section show that the performance is really harmed by the inefficiency of the fetch in most of the cases. According to the Table 3.1 we can see that prediction accuracy varies from 84.11% through 99.36% which is in average a good accuracy for the prediction.

Figure 3.5 shows the factors which caused the loss. Each one of the four causes are presented for each benchmark in the proportion in which each occurred during the simulation. The black bar represents mispredictions while the dark gray represents misfetches. The light gray represents i-cache misses and the hatching bar represents the fetch buffer full.

For 66% of all benchmarks mispredictions really dominate and represent the main cause of loss to the fetch. For other benchmarks such as *Fpppp* and *Vortex* the main cause of loss is the stall created by series of misses in the instruction cache. And, for *Applu*, *Apsi*, *Mgrid* and *Turb3d* benchmarks we still see that the fetch is mainly limited by the buffer which is frequently full preventing the fetch to work.

Table 3.1: Prediction Accuracy Rate

Benchmark	Dynamic Branches (%)	Accuracy (%)	Misprediction Rate (%)
Compress	**22.10**	84.94	**15.06**
Gcc	19.22	92.06	**7.94**
Go	14.95	84.11	**15.89**
Ijpeg	10.80	89.97	**10.03**
Li	**22.80**	94.87	5.13
M88ksim	**28.15**	97.06	2.94
Perl	19.38	**98.93**	1.07
Vortex	15.96	97.58	2.42
Applu	*2.53*	84.73	**15.27**
Apsi	*3.88*	98.73	1.27
Fpppp	*1.72*	93.48	6.52
Hydro2d	19.43	**99.14**	0.86
Mgrid	*1.32*	97.79	2.21
Su2cor	**21.81**	**98.80**	1.20
Swim	17.34	**99.36**	0.64
Tomcatv	**20.55**	**99.07**	0.93
Turb3d	*6.65*	94.17	5.83
Wave5	15.53	98.72	1.28
Average			
int	19.17	92.44	7.56
fp	11.07	96.40	3.60
int & fp	14.66	94.64	5.36

The occurrence of fetch buffer full can be caused by high latency functional units, for example. Also misses on the data cache can delay the execution of loads and stores which consequently delay the execution of other instructions in the dependency chain.

Many causes may increase the occupancy of the fetch buffer. We did not perform simulations to verify what are the causes that affect the fetch buffer. However, for the cases studied we can say that the fetch is not a bottleneck when the stall is due to a fetch buffer full.

A very small fetch buffer may cause directly the fetch to stall but in the cases studied we used a 256 entries fetch buffer which is considered wide enough to handle many outstanding instructions. In the cases studied we observed buffer full, basically, in float point benchmarks. This occurs because float point instructions take more time to execute. That is, they have higher latencies the integer instructions.

3.3.3 Effective Fetch

Subtracting the lost cycles from the total execution cycles we have the total number of cycles where at least one useful instruction was fetched. Figure 3.6 shows the percentage of total cycles where there was an effective fetch.

In this case we are not measuring bandwidth utilization. This means that if at least one useful instruction is brought from the cache then this is considered an effective fetch. If there is a stall or instructions from the wrong path are brought from the cache then it is considered a useless fetch cycle.

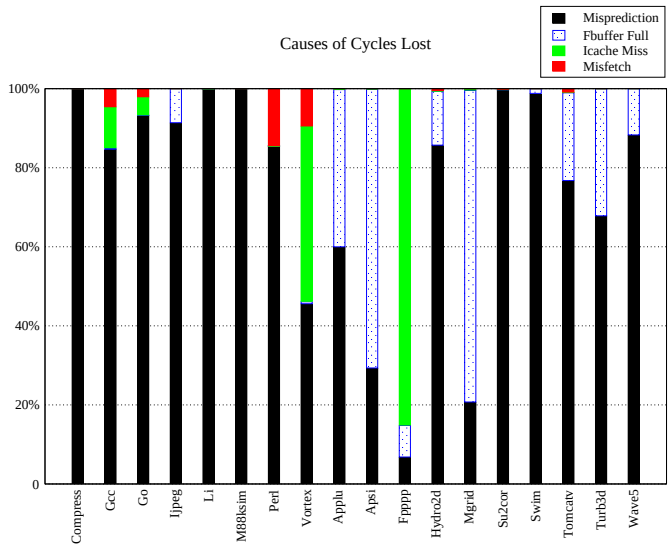

Figure 3.5: Causes of Loss

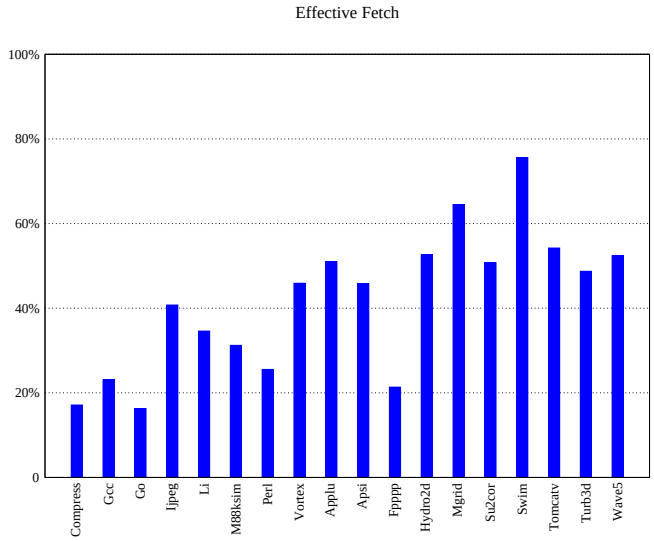

Figure 3.6: Effective Fetch

Among the total time of execution we can see that the fetch unit is in average being effectively utilized by 41.74% of total time. In the worst case, i.e. *Go* benchmark, the fetch is effectively used in only 17.22% of the total time. For the best case, i.e. *Swim* benchmark, the fetch effectively brought instructions into the pipeline in 75.6% of total cycles.

The graph on Figure 3.6 justifies the role of the fetch unit on the performance of such an advanced superscalar machine. Even with a relatively accurate branch predictor the performance of the fetch unit is heavily limited by mispredictions especially in integer benchmarks.

3.4 Impact on Efficiency

We gave two different insights into the fetch performance in the last two sections. Firstly, we measured and analyzed the stalls caused in the fetch unit by several factors. Secondly, we looked at the loss caused indirectly by the occurrence of mispredictions and misfetch associated with the stalls caused by other factors. At the end, we showed the impact caused be these factors on the fetch effectiveness.

In this section, we are looking into the impact caused by mispredictions on the efficiency of the machine. Considering the same simulations, we compared the number of committed instructions against the number of executed instructions. Due to mispredicted branches, the number of instructions executed is greater than the number of instructions committed.

When a branch is mispredicted the processor executes instructions down the wrong path. That can incur several factors which undoubtedly harm the performance. Instructions executed and later discarded consume power as well as resources. When a resource is allocated for an invalid instruction the resource operates normally. Therefore, other instructions can be delayed in the execution pipeline because an invalid instruction is taking the place of a valid one.

For example, instructions from the wrong path can access the memory generating misses, which implies in a high number of cycles waiting for useless data. As an instruction issued from the wrong path can only be invalidated after the branch is resolved many outstanding misses may interfere the execution of other valid instructions whereas valid and invalid instructions are competing for the same set of resources. Hence, there are two negative points in executing instructions from the wrong path. First, it takes power and resources. Second, these instructions may interfere in the execution of valid instructions imposing extra delays.

In our reference architecture a misprediction causes a minimum of 11 cycles of penalty, if the branch takes only one cycle at each stage and the misprediction is detected as soon as the condition is evaluated. Sometimes a branch may take more time in one stage of the pipeline because of a true data dependency or other resource constraints. In these situations the penalty is even greater because the condition will take more time to be evaluated, allowing the pipeline to process even more instructions from the wrong path.

The problem is similar when misfetches occur. The difference resides in the fact that a misfetch can be detected earlier than a misprediction. As soon as the branch is decoded and the target address is known the processor can identify an on going misfetch and so the penalty is smaller. In the reference architecture used here there is a 4 cycles minimum penalty to recovery from a misfetch.

The prediction of a branch is a compound of two things. First, it is necessary to predict the direction of the branch and second, it is necessary to obtain the target address. If the branch is predicted not taken, then the target address is the sequential. If the branch is predicted taken, the procedure is different. If the branch hits on the BTB, the address found is used to fetch the next instruction. Until the branch is decoded, that address is assumed to be correct. However, another problem may happen in this stage. Conflicting branches can store their target addresses at the same location, and so, it is possible that even with a hit in the BTB the address stored in the location accessed pertains to another branch.

Notice that if a misfetch is detected and recovered there is still the chance that the direction has been mispredicted and thus misprediction penalty will be charged as well.

3.4.1 Overall Statistics

Tables 3.2 and 3.3 summarizes some of the results presented in this chapter. They are presenting information regarding the performance of the fetch and some other resources in the architecture simulated.

The benchmarks are in the first column. Data cache miss and Instruction cache miss rates are on the second and third columns respectively. In the fourth column there is the percentage of committed branches and on the fifth column the misprediction rate for the respective benchmark. In the sixth column we present the percentage of mispredicted branches among the total number of instructions committed. In columns 7th, 8th and 9th we present respectively misprediction penalty, stalls and total loss. Total loss so called fetch loss includes misprediction penalties and stalls. In the tenth column we present the ratio between the number of committed instructions and the number of executed instructions. On the last column we show the IPC.

We sorted the benchmarks in a manner to expose the effect of mispredictions on the performance. The benchmarks are disposed in the table through an ascendant order of misprediction penalties, i.e. the first benchmark has the lowest misprediction penalty while the last benchmark has the highest misprediction penalty.

Table 3.2: Integer Overall Statistics

Benchmark	Dmiss	Imiss	Comm Br %	Mispred %	Insn Mispred %	Mispred penalty %	Stall %	Fetch Loss %	Commited/Executed %	IPC
Vortex	0.0082	0.0103	15.88	2.42	0.38	24.62	26.78	54.12	88.79	3.87
Ijpeg	0.0019	0.0000	7.99	10.03	0.80	54.09	8.80	59.26	74.59	4.16
Perl	0.0000	0.0000	19.04	1.07	0.20	63.63	5.81	74.47	59.83	2.85
Gcc	0.0066	0.0038	19.93	7.94	1.58	65.09	13.33	76.96	50.52	2.02
Li	0.0073	0.0000	22.82	5.13	1.17	65.35	4.60	65.41	59.83	2.48
M88ksim	0.0060	0.0000	22.96	2.94	0.68	68.79	4.05	68.79	55.77	2.01
Go	0.0041	0.0015	14.86	15.89	2.36	78.02	8.88	83.72	35.98	1.72
Compress	0.0060	0.0000	19.63	15.06	2.96	82.89	4.01	82.89	29.82	1.29
Average	0.0050	0.0020	17.89	7.56	1.27	62.81	9.53	70.7	56.89	2.55

Table 3.2 presents statistics extracted from the simulation of the SPEC95int benchmarks. It is very clear the impact of mispredictions on performance of integer benchmarks. In average 17.89% of committed instructions are branches and 7.56% of those branches are mispredicted. This leads to an average of 1.27% of total instructions causing misprediction penalties.

In spite of only 1.27% of total instructions cause mispredictions, the penalties correspond to an average of 62.81% of total execution cycles. Although the average performance of the branch predictor for these benchmarks is around 93% of accuracy a severe penalty is still incurred.

We also observe that as the misprediction penalties increase (top to bottom) the ratio of committed instructions decrease as well as the IPC. As expected, there is an inverse relation between the misprediction penalty and the performance. The more misprediction penalties paid the less the performance.

An interesting situation occurred for *M88ksim* benchmark. The benchmark presented one of the lowest occurrence of mispredictions, 0.68% of total instructions. *Li* benchmark has almost twice more mispredictions where 1.17% of total instructions were mispredicted branches. However, the misprediction penalty paid by *M88ksim* benchmark is higher than for *Li* benchmark. This is the situation we mentioned previously. Even with lower number of mispredictions the *M88ksim* benchmark suffer higher penalties. This may be due to data

dependencies that delays the execution of the conditions, delaying the branch resolution and the misprediction detection.

Despite be the third highest misprediction rate, *Ijpeg* benchmark has the highest IPC. We can see that this is due to the low frequency of branches committed, less than 50% of the average frequency of branches for the integer benchmarks. Furthermore, it presents a better performance than *Vortex* benchmark even presenting a higher misprediction rate and misprediction penalty. In this case, *Vortex* benchmark presented the highest instruction cache miss rate accounting for the highest fetch stall percentage among all integer benchmarks.

Notice that in average only 56.89% of executed instructions are really committed or, in other words, are useful. This is because the fetch is inefficient even presenting an average of only 7.56% misprediction rate and 0.2% Icache miss rate.

Table 3.3: Float Point Overall Statistics

Benchmark	Dmiss	Imiss	Comm Br %	Mispred %	Insn Mispred %	Mispred penalty %	Stall %	Fetch Loss %	Commited/Executed %	IPC
Fpppp	0.0002	0.0574	1.44	6.52	0.09	5.28	73.56	78.66	93.36	1.98
Mgrid	0.0172	0.0000	1.29	2.21	0.03	7.33	28.39	35.52	99.09	7.51
Apsi	0.0390	0.0000	3.86	1.27	0.05	15.85	38.62	54.22	97.92	4.88
Swim	0.0008	0.0000	17.96	0.64	0.11	24.08	1.48	24.40	86.76	4.38
Applu	0.0193	0.0000	2.19	15.27	0.33	29.36	21.79	49.04	89.55	6.38
Turb3d	0.0100	0.0000	5.3	5.83	0.31	34.73	18.15	51.28	87.01	4.51
Tomcatv	0.0001	0.0000	21.33	0.93	0.20	35.08	11.77	45.78	82.42	3.89
Hydro2d	0.0002	0.0000	21.71	0.86	0.19	40.56	8.49	47.38	70.91	3.89
Wave5	0.0042	0.0000	16.12	1.28	0.21	41.98	7.09	47.60	87.71	3.53
Su2cor	0.0002	0.0000	23.83	1.20	0.29	49.11	2.55	49.25	65.61	3.75
Average	0.0091	0.0057	11.5	3.6	0.18	28.34	21.19	48.31	86.03	4.47

Table 3.3 presents statistics extracted from the simulation of the SPEC95fp benchmarks. In average, 11.5% of committed instructions are branches and 3.6% of those branches are mispredicted. This leads to an average of 0.18% of total instructions causing misprediction penalties.

Notice that on average, float point benchmarks have a higher IPC than integer benchmarks. In the experiments performed, float point benchmarks achieved an average IPC of 4.47 instructions per cycle where integer benchmarks achieved only 2.55.

Another interesting point is that for integer benchmarks the performance of the branch predictor was worst when the number of committed branches increased. The opposite situation was true for float point benchmarks and the performance of the predictor was better for those benchmarks with lowest frequency of branches.

Despite this interesting controversial analysis the performance of the float point benchmarks is limited by mispredictions and Icache misses too. We sorted the benchmarks accordingly with the misprediction penalty, Table 3.3. The lowest misprediction penalty was reported for *Fpppp* benchmark, with only 5.28% of cycles being wasted due to mispredictions. However, this benchmark presented a very high Icache miss rate, almost 6%, leading to a very high occurrence of fetch stalls. By the way this benchmark reported the highest fetch stall rate among all benchmarks. The rest of the float point benchmarks did not presented significant Icache miss rates.

Figure 3.7 presents instructions committed *versus* instructions executed. The goal is to show the pollution introduced by mispredictions. The inefficiency of the fetch unit on bringing instructions from the wrong path of mispredicted branches causes an immense pollution.

In extreme bad cases such as *Compress* benchmark only 29.82% of instructions executed are finally committed. This represents a major penalty not only by squashing useless

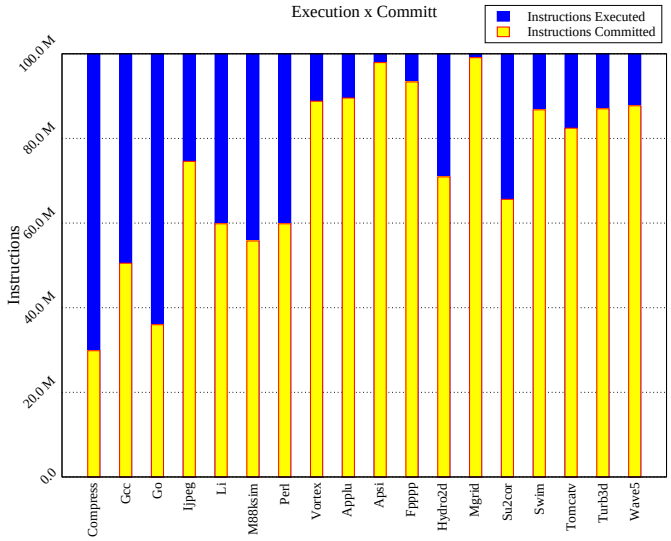

Figure 3.7: Machine Efficiency

instructions, but by using resources to execute them as well. In the best case, on the other hand, we see 99.09% of instructions executed becoming committed. This is the case of benchmark *Mgrid* which presented the highest IPC.

In average, only 56.89% of executed instructions are committed, when executing integer benchmarks. In this case, the machine is being underutilized and its efficiency is around 50% only. A different behavior is shown by float point benchmarks. In average, 86.03% of instructions are committed which shows the reason to have an average IPC as twice high as presented by integer benchmarks.

3.5 Perfect Branch Prediction and its Impact on Performance

Figure 3.8 compares the IPC achieved using a perfect branch predictor (so called ideal performance) and using a conventional branch predictor. The dark gray bar shows the performance when applying the perfect predictor while the light gray bar shows the performance with the conventional predictor.

For all integer benchmarks the impact on performance is considerable. In all cases except *Vortex* the performance of the conventional machine did not achieve 50% of the ideal performance achieved by the machine with perfect predictor.

Vortex presented the lowest misprediction penalty accounting for only 24.62% of total cycles. Even though *Vortex* presented the highest Icache miss rate among the integer benchmarks, its performance was really close to the performance obtained by the ideal architecture which means that limitations where not imposed by the occurrence of branches but for other factors.

In the float point scenario, the limitation is mainly imposed by data dependencies and resource conflicts. In this set of experiments we did not modify any feature from one architecture to another but the predictor. Both architectures have the same limitations and all mechanisms applied are exactly the same except the predictor.

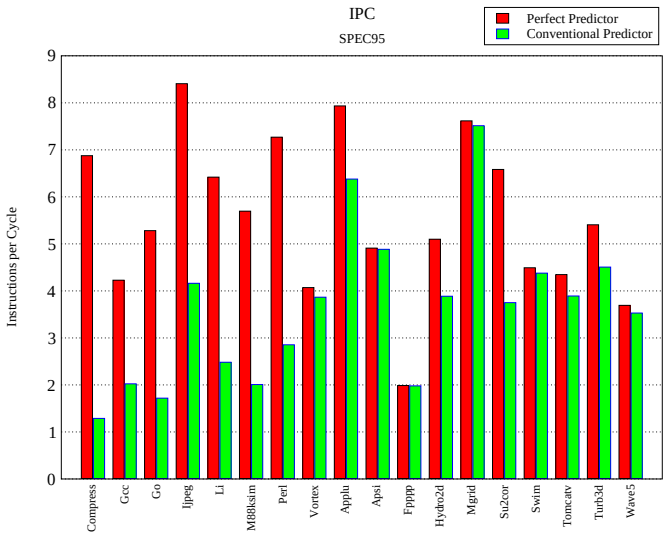

Figure 3.8: Machine Perfomances with Perfect Prediction and Conventional Prediction

Benchmarks *Applu, Hydro2d, Su2cor* and *Turb3d* presented relative worst performance than the ideal machine. Although the worst case represented by *Su2cor* achieved more than 50% of the ideal performance.

We remark though that the performance of float point benchmarks is mainly affected by other factors such as data dependencies and resource conflicts. On the other hand, for integer benchmarks, misprediction reduction is crucial to obtain good performance.

3.6 Branch Prediction and Fetch Bandwidth

Delivering more instructions to the execution engine is certainly critical. Conventional fetch schemes based on instruction cache and branch predictor are constrained by branches [LEE84, YEH91, LAM92, WAL93] and can only delivery up to one basic block of useful instructions to the execution engine, if the prediction is correct.

Fetch bandwidth relies on cache hit rate, accuracy of branch predictor, taken branches, cache alignment and frequency of branches or average size of basic block.

However, the frequency in which branches are found, in integer code specially, imposes extra limits. Even if the predictor successfully predicts the branch, a taken branch disrupts the natural flow of instructions, as shown in Figure 3.9. Either reducing/eliminating taken branches or predicting multiples branches per cycle is crucial to increase fetch bandwidth.

Several schemes have been proposed to reduce the negative effect of taken branches [UHT97]. An effective technique to increase fetch bandwidth must: (i) predict branches successfully and (ii) fetch multiple basic blocks per cycle.

3.6.1 Branch Prediction Techniques

Branches introduce extra time in the execution if their outcome is not known. Since branch prediction is available, while a branch is being resolved, instructions from the

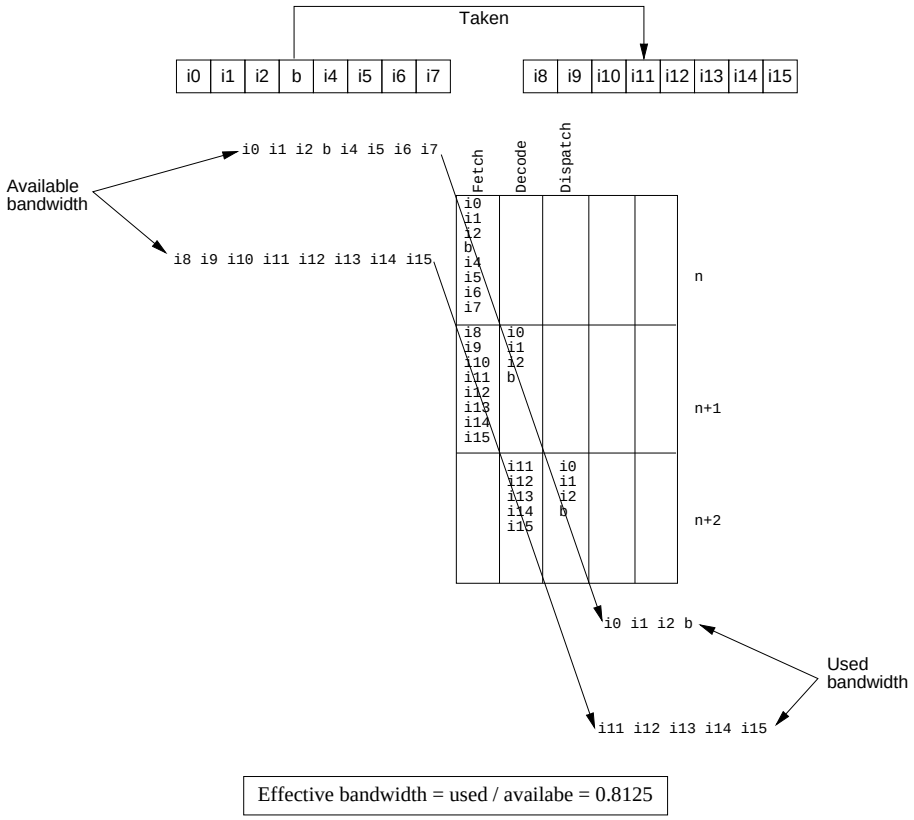

Figure 3.9: Instruction Flow Disruption

predicted path can be executed speculatively.

There are basically two forms of branch prediction: *static* and *dynamic*. Static branch prediction determines the predicted outcome of a branch prior to run-time and the outcome remains unchanged throughout the program's execution. On the other hand, dynamic branch prediction schemes gather information about the outcome of past branches and use that information to predict the behaviour of next branches.

Currently, dynamic schemes are widely used at the expense of a more complex and costly hardware. However, dynamic branch prediction schemes are better able to adapt to changes in behaviour than static schemes [TAL95].

If the prediction succeeds, whether the predictor is static or not, the branch's resolution time has been successfully hidden with useful computation. If the prediction fails, the speculative instructions must be squashed, but otherwise no harm has been done. With prediction, branches only introduce stall time on a misprediction [SKA99].

Dynamic schemes base the prediction on the history of past branches. In the literature it is possible to find three different basic types of predictors. Accordingly with [SKA99] they are: bimodal, two-level and hybrid.

Bimodal predictors capture the outcome of branches and store them in a multiple entry table, one per branch. Each entry of the table stores a binary value which indicates whether the branch will be taken or not in the next time it executes. A more sophisticated bimodal predictor was proposed by Smith [SMI81] where a two-bit saturating counter was used instead of a simple binary value.

The two-bit saturating counter allows to reduce the mispredictions when a branch change its direction from taken to not-taken and vice-versa. The conceptual advantage is based on the sense that the prediction only changes if the branch changed its direction twice consecutively. Mispredictions are reduced by using the two-bit counter and average accuracy rate of 76%-92% was reported by Smith in his studies.

Yeh and Patt [YEH91] proposed the two-level adaptive branch predictor. The key factor was to keep track of branch history patterns. Instead of predict the outcome of a branch based only on the previous outcomes of the same branch, the two-level adaptive predictor predicts the outcome of a branch based on the outcome history of preceding branches.

The two-level adaptive branch predictor shows a very low misprediction rate of 3.7% for a 128K-bit table, according with reports. The counters are used to track the history pattern of branches, and not the overall behaviour of individual branches.

The history can be local or global. While local history allows to keep track of similar pattern branches, the global history allows branches to easily see the behaviour of other recent branches. This is providential for predicting sequences of correlated branches [PAN92, YEH92, YEH93a].

If the branch prediction table is not sufficiently large, two branches may share the same entry. This may also happen if the two branches share the same history. Even though the amount of interference is low, it results in an average increase in the number of mispredicted conditional branches of 41% as showed by Talcott et al. [TAL95].

The aliasing problem can be alleviated by combining the history bits with some bits from the branch's address. McFarling [MCF93] proposed to XOR the two patterns together in a way that branches that share the same history can be distinguished by their addresses.

As explained previously, local and global history are used to predict different types of behaviours, i.e. branches. As within programs some branches are best predicted by using a local history predictor, while others are best predicted by using global history branch predictors, the idea of hybrid schemes is to combine both types of predictors with a selector which determines which prediction is better for a given branch [MCF93, CHA95, EVE96].

An example was described by Kessler [KES99]. He described the predictor used in the Alpha 21264 [GWE96]. The processor employs a hybrid scheme which selects a prediction from a local and global predictors achieving from 90% to 100% accuracy in some applications. Seznec also proposed a "De-aliased" global predictor which achieves the same prediction accuracy level of *gshare* or *gselect* [YEH93a] using less than half of the transistor budget [SEZ99].

3.6.2 Increasing the Fetch Bandwidth

In general, by multiporting the instruction cache and the BTB generating multiple fetch addresses and branch predictions per cycle, fetch schemes are able to overcome the single fetch block bottleneck [YEH93b].

Conte et al. [CON95] proposed a scheme called *Collapsing Buffer*. The idea is to remove useless instructions between an intrablock branch and its target. That merging technique allows the target instruction to follow the branch in the decoder, (Figure 3.10). The scheme was also extended to allow interblock branches to be followed by their targets. That extension suggests an interleaved instruction cache, an interleaved branch target buffer, a multiple branch predictor, and an interchange and alignment network.

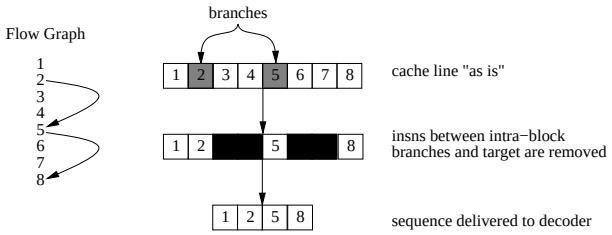

Figure 3.10: Collapsing Buffer Scheme

Yeh et al. [YEH93b] presented the Branch Address Cache (BAC) scheme. That scheme is similar to the collapsing buffer in the sense that there is also an interleaved instruction cache, a multiple branch predictor, an interleaved interchange and alignment network and a branch address cache. The later contains up to 14 basic blocks addresses when up to three branches can be predicted at a cycle. From these, three basic block addresses corresponding to the predicted path are selected.

Essentially, these two schemes [CON95, YEH93b] allow to predict multiple branches and fetch non-contiguous basic blocks from a multiple port instruction cache. Nevertheless, because instructions are placed in the instruction cache in their original (static) order, fetch through branches is not trivial, since code is not requested accordingly in the static order but in the dynamic order.

Rotenberg et al. [ROT96] presented a scheme to store dynamic sequences of instructions in the form of dynamic traces. The Trace Cache (TC) is able to capture these dynamic sequences according with the predictions or the outcome of recently executed branches and store them into the trace cache. The traces are then accessed from the trace cache increasing the bandwidth utilization because in fact the "taken branch disrupt effect" is reduced by storing instructions continuously in the dynamic order.

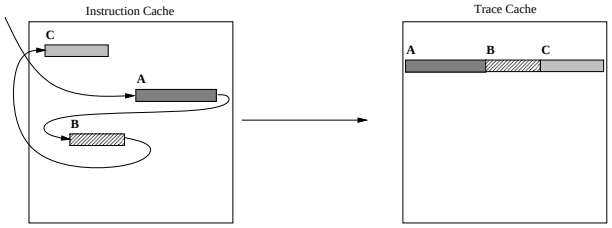

Figure 3.11: Dynamic Trace and Trace Cache

Figure 3.11 shows three basic blocks (A, B, C) in the instruction cache. Blocks are not sequentially nor contiguously stored in the i-cache. The trace cache fill unit detects this dynamic behavior and stores, sequentially, the three basic blocks in a single trace cache line. This way, it allows to fetch in a single cycle all basic blocks if the prediction for all intermediate branches is the same.

Friendly et al. [FRI97] proposed alternative fetch and issue policies to the trace cache fetch. Essentially they studied the effects of partial matching when the predictor requests a sequence of blocks (trace) which is not completely stored in the trace cache. Or, in other cases, the trace stored does not match exactly the predictor's sequence but only a partial match is observed. In this case, the predictor selects which blocks from the trace will be issued to the core.

Furthermore, they also propose a technique called inactive issue. The sense of this technique is to issue also the blocks of the trace that do not match exactly the prediction. That is, with the partial matching technique alone, the blocks that do not match the predictor are discarded. With inactive issue, all blocks within a trace are issued whether they match or not the prediction made.

The blocks that do not match the prediction are issued inactively and the changes they make to the register table are not considered valid for subsequent issue cycles. If the prediction made was correct the inactive instructions are discarded. If the prediction was incorrect, the processor has already fetched, issued and possibly executed some of the instructions along the correct path [FRI97]. The partial matching technique improves performance of the SPECint95 benchmarks by an average of 12% over a trace cache not implementing partial matching. Adding the inactive issue on top of partial matching the average improvement reaches its 15%.

Patel et al. [PAT98] proposed the combination of two techniques to improve the effectiveness of the trace cache approach. They proposed to promote some branches in order to alleviate the pressure on the branch predictor, reducing the interference of some easy to predict branches over the overall performance of the predictor. Branch promotion dynamically converts strong biased branches into statically predicted branches. Branch promotion allows to increase the number of available instructions to be packed into a trace by reducing the number of branches and increasing the predictor's bandwidth.

Another technique called trace packing was employed together with branch promotion in order to pack as many instructions as possible into a trace cache line. When applied together, both techniques allowed an increase in 17% of the effective fetch rate over a trace cache which used neither. However, a small 4% improvement in performance was observed, even decreasing the number of mispredicted branches and increasing the effective fetch rate. Moreover, promotion and packing lose performance potential due to an increase in branch resolution time. An architecture with ideal memory scheduling realized an improvement of 11% according with the studies presented by [PAT98].

Friendly et al. [FRI98] proposed to add function to the fill unit which is responsible to collect blocks of instructions and combine them into traces to be stored within the trace cache. The fill unit applied dynamic optimizations to the sequence of instructions collected, before to store them into the trace cache.

The techniques exploited the latency tolerance of the fill unit because it is not on the critical path. Four types of optimizations were studied and the performance gain showed an average improvement of more than 17% for the SPECint95 benchmarks. Another advantage is that the fill unit can perform multi-cycle operations without affecting the performance and by combining multiple blocks of instructions from a single path of execution it can easily perform optimizations across the basic blocks boundaries.

In a follow-on work, Rotenberg et al. evaluated different alternatives for the trace cache microarchitecture. Trace caches provide the capability of fetching past multiple, possible taken branches without the complexity and latency of equivalent bandwidth instruction cache designs [ROT99a]. The authors showed that the trace cache can improve from 15% to 35% over an equally-sophisticated but contiguous multiple block fetch mechanism.

Furthermore, authors summarized their experiments with some major conclusions. They detected that longer traces improve trace prediction accuracy and that the overall performance is sensitive to the size and associativity of the trace cache. However, the cost is on the redundant instructions storage.

Larriba et al. [RAM00] analyzed the level of redundancy generated by a conventional trace cache where traces are not selectively stored. The work is based on the assumption that some traces may contain sequences of instructions that are already stored in the instruction cache. This is particularly true when branches are not taken, hence the same sequences stored in the trace cache are also stored in the instruction cache, for example. They proposed a selective trace storage to avoid trace redundancy between the trace cache and the instruction cache. A modification introduced to the fill unit allowed the trace cache to store only those traces containing taken branches which cannot be obtained in a single-cycle from the instruction cache. The results showed that selective trace storage and the trace cache software, employed with a 2KB trace cache, performed as good as a 128KB trace cache without selection.

A decoupled preconstruction mechanism was proposed by Jacobson [JAC00] to reduce compulsory and capacity trace cache misses. The idea is that the preconstruction mechanism observes the dispatch stream in order to predict the future paths to be followed. In that way, the preconstruction mechanism is able to fetch static instructions from the predicted future region of the program and constructs a set of traces in advance, similar to a prefetch scheme.

The trace preconstruction presented by Jacobson reduced the trace cache miss rates from 30% to 80% for SPECint95 benchmarks. An overall performance improvement of 3% to 10% was reported with preconstruction.

Seznec et al. [SEZ97] also proposed a mechanism to fetch multiple non-consecutive blocks. Information from the current instruction block was used to predict the block following the next instruction block instead of use it to predict the address of the next instructions or block as usual.

Michaud et al. [MIC98] compared Pros and Cons of TBA (*Two-Block Ahead*) against the Trace Cache [ROT96]. The authors showed that TC outperforms TBA in many aspects from providing more fetch bandwidth to presenting smaller misprediction penalties due to a less complex pipeline.

On the other hand, TBA outperformed TC for small cache sizes. Accordingly with the authors, this was due to the poor TC hit ratio. Both schemes rely mainly on the branch prediction to decide which blocks to fetch or to decide which blocks to concatenate and store in the trace cache. Hence, there is the need for more accurate multiple branch predictors to get more benefits from both schemes.

Again Michaud et al. [MIC99] proposed an extension of the Two-block Ahead Predictor: the Extended Two-block Ahead Predictor (E-TBA). The scheme allows to fetch 4 basic blocks in a single cycle when the first and third branches are both taken, which restricts the scheme. They compared the latter against previous schemes: (One Block Ahead predictor (OBA), Extended One Block Ahead predictor (E-OBA) and Two block Ahead Predictor (TBA).

They demonstrate that the available parallelism in an instruction window grows approximately as the square root of its size. The instruction parallelism extractable from a program grows as the square root of the inverse of the misprediction rate. They analytically prove that there is a very low performance benefit of increasing the fetch rate over a threshold also proportional to the square root of the distance (in instructions) between two consecutive mispredictions.

To reduce the branch misprediction problem other alternatives can be pursued. A compiler based approach such as predication can convert control dependencies into data

dependencies. When the correct branches are chosen the benefit of a larger pool of valid instructions overcomes the addition of new data dependencies [MIC99, ?].

A second alternative is to execute both paths of a conditional branch rather than predicted the most likely taken path. Some schemes fork both paths of every conditional branch at the expense of a exponentially growth as more branches are very likely found in each new path created [UHT95]. Other approaches fork both paths of only certain branches [AHU98, HEI96, KLA98a] based on a confidence predictor [JAC96]. The fetch engine is a bottleneck in such approaches because multiple paths must be fetched at the same cycle to mimic a perfect branch predictor and improve fetch bandwidth.

4. Branch Misprediction and Multipath

As presented previously most of the current research to increase instruction supply rely on branch prediction [LEE84]. Their intention is basically allow more than one basic block follow to the decoder thus increasing instruction throughput. Obviously, performance depends on accuracy of the predictor since instructions belonging to predicted paths are usually executed speculatively. If the prediction is not correct a misprediction penalty is paid.

Misprediction penalty has many effects on the performance. The deeper the pipeline the greater the penalty. As the penalty is calculated as the time between the mispredicted branch has been detected and the correct instruction enters the pipeline, so as more stages are added more time is required to fill again the pipeline with useful instructions.

The performance of current branch predictors suggest an accuracy rate of about 95% or more. Despite that great performance, few mispredictions are enough to harm the performance, because the penalty is still very high.

Current predictors are constantly increasing their accuracy by increasing the size of the branch table or combining different predictors in order to try to capture the behaviour of different behaved branches. The success of a prediction usually depends on the history of such a branch or sometimes on the behavior of other (correlated) branches. Unfortunately, some branches are hard-to-predict imposing more difficulties to predictors get to 100% accuracy.

One interesting form of reducing the probabilities of misprediction is multipath. Multipath has been studied but always deemed as very expensive because of its nature of replicate too many resources. To achieve the same performance as a machine with an oracle predictor (i.e. no instructions have to wait for branches to be resolved and the outcome is always known) a conventional machine must execute all possible paths through a program [LAM92].

Pursuing multiple paths may be expensive but it is the only way to eliminate misprediction penalty associated with the execution of instructions belonging to the wrong path of a branch.

4.1 Following Multiple Paths

4.1.1 Eager and Disjoint Eager Execution

Various proposals have been made concerning the execution of multiple paths. The idea seems to be very attractive in the way that it can potentially eliminate the cost (in cycles) due to mispredictions. As this cost is very high, eliminate it completely is undoubtedly a very attractive feature which compounds this paradigm.

Uht presented one of the most classical works on multipath. For reducing the negative branch effects and minimize the dependencies he proposed DEE - Disjoint Eager Execution [UHT95]. DEE is based on the idea of executing multiple paths simultaneously in order to avoid completely or partially mispredictions and misspeculations. Uht presented basically two ideas around which he designed two models of multipath. The two models are:

Eager Execution is a full aggressive multipath. The fetch follows down both paths of every branch encountered. The penalty associated with mispredictions is nullified once both paths are fetched and executed. When the outcome of a branch is known the incorrect path

is squashed. The problem is that this model grows exponentially with the outstanding branches. The good thing is that no branch prediction needs to be made. Therefore, the amount of resources needed to implement the Eager model may turn the implementation unfeasible.

DEE - Disjoint Eager Execution. DEE is a selective multipath model where only the most likelihood branch paths are provided with resources. DEE is based on the cumulative probability of a branch path is executed. Instead of fetch and execute both paths of a branch DEE follows only the most probably branch paths thus reducing the amount of resources needed. Resources are assigned to paths accordingly to their cumulative probability or likelihood of execution. The DEE exhibits better performance than a single path architecture and Eager with constrained resources without eager execution's high cost [UHT95].

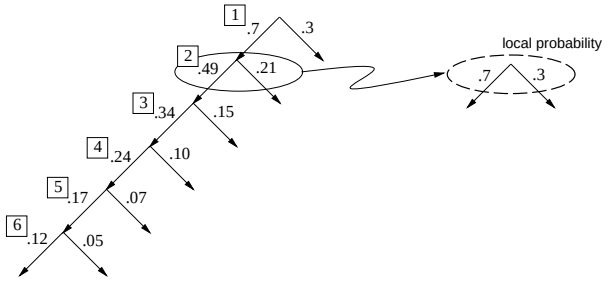

Figure 4.1: Single Path Architecture with Cumulative Branch Probabilities

Figure 4.1 shows an example of several branch paths along with their cumulative probabilities. The example follows an arbitrarily fixed probability of .7 (70%) and .3 (30%) for each path, respectively left and right paths. For each branch there are two possible paths (left and right arrows) with local probabilities (70% and 30%). Along the tree the probabilities are calculated based on the predecessor branches. Thus the probabilities along the dynamic tree are cumulative. As the branches are resolved the probabilities must be recalculated to reflect the result of the oldest branch. Each time a branch is resolved a new probability for each outstanding branch is calculated. This can be done by assuming the local probabilities are known by the time the branch is encountered and for each resolved branch a new set of probabilities must be re-calculated for all outstanding branches.

The numbered squares represents the order in which each path would be pursued in a single path architecture respecting the cumulative probabilities depicted in figure 4.1.

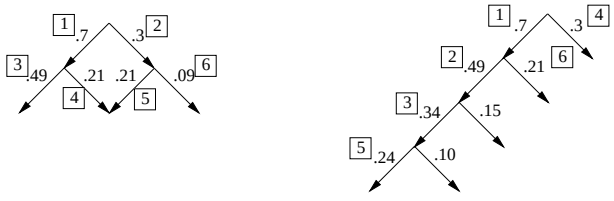

Figure 4.2: Eager and DEE Multiple Path Architectures

Figure 4.2 shows the dynamic tree now for the Eager and DEE models presented by Uht. In the left-most figure we see the Eager model. In the Eager execution, assuming 6 paths can be handled simultaneously, the order in which each path would be considered

follows the square numbered from 1 through 6. The difference from figure 4.1 is that the execution (i.e. the architecture) is fully divided among all paths for every branch, i.e. for each encountered branch the execution goes down both path does not matter the probability of each path to be taken or not.

In the right-most side of the figure 4.2 we see the DEE model. DEE is selective in the way that paths are selected based on their cumulative probability. So the order in which 6 paths would be considered would follow the dynamic probabilities for all outstanding branches instead of simply executed both paths of every branch as done in the Eager model.

The difference between the DEE and a single path is that in the first only one path of each branch is executed depending on the prediction, in this case the probability. In the Eager, a given branch may have both paths executed since their probabilities are the highest among all outstanding (fetched but not executed) branches. For example, for the first and second branches in the DEE tree we have both paths executed. But the least likely path of the first branch is only pursued after the most likely path of the fourth branch (step 5) is pursued. After that, the most likely path among all outstanding paths become the right side path of the first branch. The same for the right side path of the second branch in that tree and so on.

Notice that a new branch can change all the probabilities as well as a branch that become solved at the time we have to decide which path to pursue.

In DEE it is assumed that the local probabilities are known at the time the branch is encountered and they can be updated each time a new branch is encountered or a branch is resolved.

Despite its incredible potential, DEE depends on the calculus of the probability that all preceding branches have been correctly predicted. Each branch has a probability of being incorrect. The apparatus involved in the calculation of this dynamic cumulative probabilities is critical in this scheme. The authors suggested a way to bypass this however they fixed probabilities based on a profiled analysis. Although the scheme performed well because mispredictions were reduced, fixed probabilities result in paths being executed in a fixed pattern which reduces the essence dynamics of the original model.

4.1.2 Selective Dual Path Execution

Another interesting work, presented by Heil and Smith [HEI96], suggested an alternative Selective Dual Path Execution model (SDPE). SDPE restricts the number of simultaneously executed paths to two and uses a branch prediction confidence mechanism to fork selectively only branches that are more likely to be mispredicted. The approach intents to reduce the total number of outstanding paths. Only those branches which the prediction is considered low confidence have their both paths executed.

For the benchmarks and the branch predictor simulated in their experiments a branch confidence table with 3-bit resetting counters identified 20% of the branch predictions as low confidence, and those contained 75% of all mispredictions. The SDPE also uses a forking policy to decide whether a branch has to be forked or not when two paths are already in execution. The best policy allows 50% of reduction on cycles lost due to mispredictions accounting on almost 10% of reduction of total execution time.

In their experiments an interleaved instruction cache with next line prefetching was used allowing 32 instructions from two consecutive lines to be brought from the cache in a cycle. From these 32 instructions up to 8 instructions following the current PC can

be selected. But only one branch prediction can be performed and any control transfer instruction terminates a block of fetched instructions. The fetch thus is limited to one basic block or 8 instructions per cycle for only one path. The fetch alternates between the two active paths accordingly to some heuristic, e.g. the most likely correct path deserves more fetch bandwidth.

The authors identified that mispredicted branches tend to come in clusters. They measured that 29% of the mispredicted branches are distance one and 58% of mispredicted branches are within 3 branches of the previous mispredicted branch. The distance between mispredicted branches is the number of branches that separate them. If a mispredicted branch is immediately followed by a second, the distance is one [HEI96]. Moreover, low confidence branch predictions also occur in clusters.

This can degrade the performance of SDPE in two ways. First, if a branch is mispredicted but not forked, i.e. only one path is being executed, any posterior fork will not benefit since it will be squashed when the branch is resolved. Second, if a low confidence branch is forked any other low confidence branch will not be forked since only two paths can be active at the same time in SDPE.

4.2 Multipath: Pros and Cons

So far, solutions proposed to increase fetch bandwidth were described throughout this work. Moreover, simulated results proved that branches can really harm the performance and mechanisms to revert this situation are essential to improve future microprocessor's performance.

We have shown that only predicting branches is not enough. It is necessary to fetch multiple basic blocks in order to provide sufficient number of instructions to feed the execution units of an aggressive multiple issue processor. Multipath has been presented as a good technique to reduce misprediction penalty, reduce misprediction occurrence, supply more instructions reducing instructions flow disruptions and increase the potential for good performance.

Tough multipath may provide all desirable features of a modern microprocessor, there are some problems that may drown out the benefits. There are currently two forms of multipath: (i) eager multipath and (ii) selective multipath. The eager multipath, executes instructions down both paths of all branches upon the availability of resources. The selective multipath, pursue multiple paths of only branches which are considered likely to mispredict. For the later, usually a confidence scheme is added to the branch predictor so only those branches which have low confidence predictions are considered for multipath. In this chapter, we present an overview of multipath by analysing the pros and cons of this technique.

4.2.1 An Introductory Example

Figure 4.3 is a copy of a simple piece of code written in C language found in [**?**]. A "C" *if-then-else* statement is presented in order to illustrate the advantages and disadvantages of using a multipath approach.

The *if* statement tests if I and J are equal. If so, F will hold the result of adding the variables G and H. If not, F will hold the subtraction of H from G. With this example, we have illustrated a very simple case where there is a conditional operation being taken.

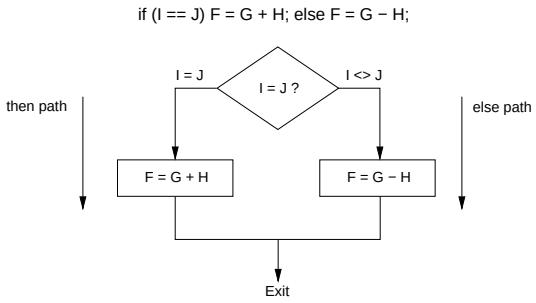

Figure 4.3: A "C" *if* Statement

Figure 4.4 shows the corresponding MIPS assembly assuming that each "C" variable (*F, G, H, I, J*) is mapped to a register (*s0, s1, s2, s3, s4*). For example, *F* is mapped to *s0*, *G* is mapped to *s1* and so on.

To simplify, we associated a letter to each instruction in the assembly code, lets say *a, b, c, d, e, f* and so on. Figure 4.5 shows how the code looks like and the position of the instructions inside an 8 instructions memory cache line.

```
        bne $s3, $s4, Else      # go to Else if I <> J
        add $s0, $s1, $s2       # F = G + H
        j Exit                  # go to Exit
Else:   sub $s0, $s1, $s2       # F = G − H
Exit:
```

Figure 4.4: The Compiled MIPS Assembly

When the cache line containing the *if-then-else* statement is fetched, the processor needs to decide whether the conditional branch (*if (I == J)*) will be taken or not in order to decide what instructions to sent to the decoder. Hence, the next dynamic instruction to follow *a* must be chosen as soon as possible in order to avoid any performance delays.

If a prediction is made, there is always the chance that the prediction turns out to be incorrect. In this case, a misprediction penalty is charged. If the prediction is correct, nothing occurs since the correct instructions are probably already being executed.

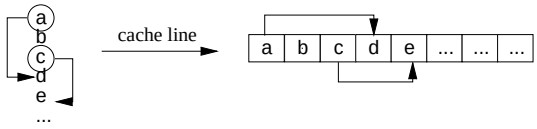

Figure 4.5: The Symbolic Example

The greatest benefit of using a multipath scheme is that since both paths are fetched and executed, there is no misprediction at all. If the eager model is employed, there is no need to predict branches since both paths are always executed. If the selective model is employed then the processor still incurs in some penalties, if multipath is not applied for a mispredicted branch.

Undoubtedly, the ideal would be to use eager multipath since mispredictions are completely eliminated. However, the eager model incurs an exponential growth in the amount of used resources.

Figure 4.6 presents the dynamic sequences of code required for executing the taken and not taken paths separately, regarding branch *a* being handled in a conventional, one branch per cycle, architecture. If the branch is not taken (left side of the tree) *a, b* and *c* can be sent to the decoder and an extra access, to the same cache line, is necessary to bring *e*. If the branch is taken, only *a* is sent to the decoder and an extra access is made to bring *d* and *e*.

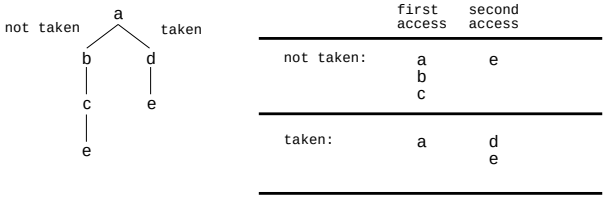

Figure 4.6: The Dynamic Sequences - Taken and not Taken

Notice that, the *extra accesses* were emphasized in order to highlight that these extra accesses are made to the *same* cache line. Two accesses to the same cache line, in two consecutive fetch cycles, is not an efficient use of resources nor an optimal form of fetching, but a simple solution to allow branch prediction.

It is necessary to fetch, to detect the branch, to predict it and then fetch the target, if the branch is taken. What turns out, is that sometimes the target is contained in the same cache line where the branch is, as shown in Figure 4.5. This problem is inherent to a conventional, non-multipath microprocessor capable of predicting only one branch per cycle.

When multipath is applied, this problem potentially grows since new paths are created and handled separately. In Figure 4.7, three accesses to the same cache line are necessary to bring instructions for both taken and not taken paths of branch *a*.

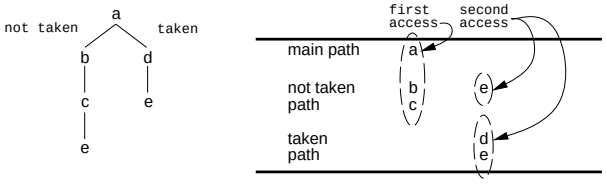

Figure 4.7: The Eager Multiple Paths Version

Another important issue comes when merge points are considered. Every conditional branch has two possible paths to follow: taken and not taken. But, after most conditional branches there is a convergent point: merge point. A merge point is the point where both taken and not taken paths converge. In the example of Figure 4.5, the merge point is *e*.

Notice that, *e* is replicated in both taken and not taken paths when multipath is applied. Since paths are handled separately, after the branch, fetch addresses are generated in separate for each path. Thus, the fetch for each path will pass through a common point (convergent) in the future.

4.2.2 An Extended Example

The examples showed previously intended to maximize the understanding of multipath issues through the analysis of a simple case. Of course, more complex situations occur in real cases, especially when series of consecutive branches are considered. Figure 4.8 extends the previous example introducing another conditional branch to the case.

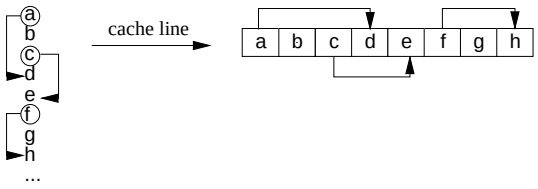

Figure 4.8: An Extended Case

The extended example has two conditional branches *a* and *f*. The number of outstanding branches and the time needed to resolve each branch determines the number of active paths necessary to achieve the desired goal which is to eliminate mispredictions. If, for any reason, a path cannot be activated, a potential situation for misprediction arises. When there are no resources available to activate both paths of a given branch, the processor must either stall or choose one of the paths to fetch and execute, creating potentially needs for prediction.

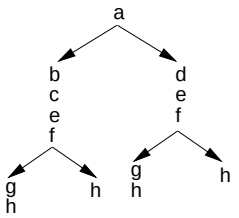

Figure 4.9: The Result: an exponential growth

Figure 4.9 shows the dynamic tree with four active paths and two outstanding branches. Note that not only the number of paths grew exponentially with the increased number of branches but also the instructions replicated throughout the tree. Instruction *e* was replicated again, as occurred in the example presented in Figure 4.7. Instruction *f* was introduced also but it represents the same situation as instruction *e* since they are consecutive with no branches between them.

However, observe that instructions *g* and *h* are replicated too. Moreover, instruction *h* appears four times in each of the active paths and instruction *g* appears in only two paths. This happened because instruction *h* is on the merged path of both conditional branches *a* and *f*.

4.2.3 New Data Dependencies

Although instructions are replicated, in fact they are not exactly the same instruction. They are dynamic instances of the same static instruction as they pertain to different data dependency chains. As instructions are brought into the pipeline through different paths, the dependency chains are different.

Lets Assume, for example, that instruction *e* is *X = F + 1*, on Figure 4.7. As *e* is on the merge point after branch *a*, then *e* is replicated in both taken and not-taken paths. Therefore, the dynamic instance of *e*, on the not-taken path, would be *X = (G - H) + 1*. And, the dynamic instance of *e*, on the taken path, would be *X = (G + H) + 1*.

There are two problems. First, *F* is written in both taken and not-taken paths. That is, an output dependency is introduced by issuing instruction from both paths of branch *a*. Second, each instance of *e* must wait for its correct copy of *F* in order to respect the true data dependency between *X* and *F*.

A register renaming technique can be applied in order to eliminate the *false* output data dependencies introduced by issuing multiple paths. But if there is an instruction on the merge point which uses the conflicting value, then each copy of the instruction must be linked to the correct predecessor in the data dependency chain.

4.2.4 Outstanding Branches and Depth

As mentioned previously, the number of outstanding branches determine the number of paths which may be necessary in order to avoid completely any misprediction, in a eager multipath architecture. That is, the number of necessary active paths depends on the latency of the previous branches.

Figure 4.10 shows an hypothetic situation where the oldest branch takes 4 cycles to be resolved. The oldest branch is fetched at cycle *n* and then it takes one cycle in each of the next four pipeline stages until it is resolved at cycle *n+4*.

Assuming that only one branch is predicted per cycle for each path, after 4 cycles the processor may have a total of 16 paths active by the time the oldest branch is resolved. That means that at cycle *n+4* the fetch unit may need 16 ports to the instruction cache in order to fetch instructions to the 16 active paths.

Furthermore, this assumption is based in a constant branch resolution time. If the oldest branch takes more than four cycles to resolve, the number of active paths may increase.

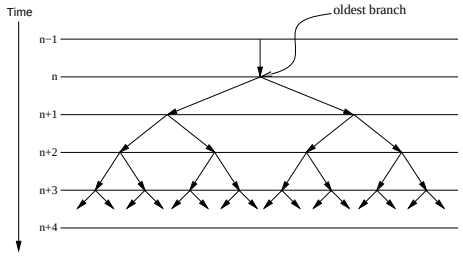

Figure 4.10: The Dynamic Tree After the Oldest Branch is Fetched

Figure 4.11 assumes that at cycle *n+4* the oldest branch is resolved. Half of the dynamic tree can be flushed at this time because there is no need to keep instructions from the wrong path since the branch is already resolved.

If the eager multipath model is used, two new paths are created each time a branch is predicted. Also, for each active path it is necessary to be able to predict at least one branch per cycle. So, the dynamic tree grows exponentially and the size depends on the resolution of the oldest branch.

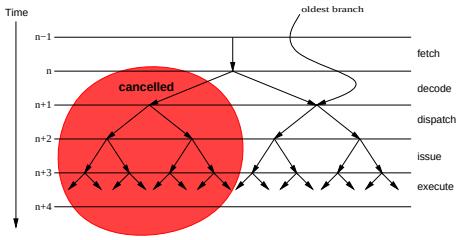

Figure 4.11: The Dynamic Tree After the Oldest Branch is Resolved

On the other hand, when the oldest branch is resolved one entire side of the dynamic tree can be squashed. Hence, the maximum number of active paths is a power of 2 of the number of cycles necessary to resolve the oldest branch. In the example presented here, the oldest branch takes four cycles to be resolved (after being fetched) so the resulting number of active paths is 16 (2^4).

4.2.5 Concluding Remarks

Essentially the most interesting benefit of using a multipath technique is to avoid mispredictions. However this major benefit may incur additional problems such as increase of necessary fetch and execution bandwidths.

The instructions fetched and issued from wrong-paths and their subordinates allow to mimic a perfect branch prediction but introduce pollution that may drown out the performance. It is well known that multipath offers an interesting form of reducing the effects of control dependency by eliminating mispredictions but persistent problems do not allow architects to consider it as a commercial alternative.

Unfortunately, the eager model has been studied but not ever implemented because of its exaggerated use of resources. The exponential consumption of resources makes this model an unfeasible option. The selective multipath, on the other hand, is most suitable way to reduce mispredictions but the performance reported previously did not motivate implementations. It is necessary still a considerable amount of resources to execute instructions from the wrong paths and the confidence estimators do not catch all mispredictions. If they did, then mispredictions could be eliminated without using a multipath technique.

The fetch bandwidth necessary to implement the eager multipath is certainly the major constraint. Also, the high number of instructions in-flight issued from wrong paths may interfere with the schedule of useful instructions. These two problems will addressed in the next sections.

5. DCE - Dynamic Conditional Execution

So many ideas have been proposed recently from executing multiple paths of conditional branches [AHU98] to executing multiple threads and processes in a single one-chip processor. All of them rely on fetching more instructions to feed starving functional units. Fetching more instructions is complicated because correct instruction addresses are completely unknown during fetch after a conditional branch. Branch prediction is applied to predict those addresses thus avoiding to stall the fetch on the presence of conditional branches. Despite the accuracy of branch predictors, the small remaining mispredictions degrade considerably the performance.

DCE is been proposed as a new architecture where it will be possible to exploit multipath with low cost implementation. Not only that, DCE is expected to diminish mispredictions and complexity at the fetch unit.

The fetch unit is a critical part of any current microprocessor. We want to make it simpler. By using DCE, we expect to allow a more natural operation of the fetch unit and take advantage from things currently unused. Looking at the operation of a conventional fetch unit, one can say that its operation is complex, expensive and extremely critical because it interacts directly with the memory and suffer the effects of unknown conditions produced by control transfer instructions, which may reduce its efficiency.

DCE is not expected to increase the complexity of the fetch unit. On the other hand, DCE is being designed to benefit from things which are already present, but unused. An example of such *"unused things"* are instructions that come into the line brought from the instruction cache, but are not used because a branch is present and its outcome is unknown. Such instructions, depending on the size of the cache line, often represent the target of the branch which is breaking the sequentially of the instruction flow.

The conventional form of fetch is driven by the execution unit and anticipated by the branch prediction mechanism. So, the cache line brought is determined by the fetch unit and instructions present into the line are selected and sent to the decoder. As the outcome of conditional branches is predicted, sometimes the selection is wrong and the instructions executed must be canceled. Current implementations already consider a recovery mechanism to fix such mistakes !

5.1 Motivation

If the a cache line contains, in such way, useful instructions, why not to execute them if they are fetched ? This was the question which motivated the start of this work. Not rarely, a big portion of the cache line is wasted because of instructions are unaligned or because there is one or more branches in the line. Therefore, bandwidth is wasted.

The point is, if an instruction exists, it should be useful somehow in the future. If not in a given moment, we presume that if the instruction is there, that is because the instruction is useful. Otherwise, the instruction should not exist.

Looking deeply into the locality of such instructions, we detected a very interesting pattern. Some branches have their targets very near, often inside the same cache line. If the taken target is inside the same line, that means that the whole not taken path is also entirely contained inside that line.

Moreover, if the taken target is inside the line, at least one instruction of the taken path is on that line and the rest is on the next sequential line, up to the next branch.

In a VLIW architecture with predicated execution this is done intentionally by the compiler. The instructions of both paths are brought automatically into the execution core but only the instructions which satisfy the condition are really executed.

DCE is similar in the sense that instructions from both paths are brought into the processor, just as a multipath or a VLIW with predicated execution. Then all instructions are executed and committed accordingly with the outcome of the non-speculative branches. Instructions that satisfy the condition are committed and the leftover is discarded.

In this way, DCE is a hybrid model where multipath is exploited more naturally than previous proposals. In DCE, a whole cache line is the unit of fetching and all instructions contained into the cache line are expected to be useful in some way.

The efficiency will depend on the agility of the fetch unit to bring the correct lines in a sequence where it is possible to have multiple paths naturally and so execute them assuring that there will be no mispredictions.

Of course, some branches have targets in far locations and cannot be executed eagerly in the same form as described above. For those branches is better to make a prediction and trust the results until the condition is evaluated.

5.2 Multipath as a form of Avoiding Mispredictions

Multipath is not a new idea. Multipath is conceptually simple but requires an exaggerate amount of resources to be implemented. Such an aggressive effort to avoid only about 3-6% [MCF93] of remaining mispredictions seems to be worthless and not marketable.

Multipath has been studied under two different approaches: eager multipath which eliminates all mispredictions while it does need very large fetch and execution bandwidths, and selective multipath which requires less resources by selecting branches which are more likely to be mispredicted, but is unable to eliminate all mispredictions.

5.3 Eager Multipath Execution

Despite the possible great benefit of eliminating all mispredictions, an eager implementation of multipath also brings new problems. First of all, the exponential growth indeed restricts the application of such an aggressive model. As for each branch two new paths are created and kept until the branch is resolved, the dynamic tree containing both paths of all branches grows until the oldest branch is resolved. At this time, the side which contains the wrong path is squashed. Hence, the size (that is, the number of active paths) of the dynamic tree depends on the duration (latency) of the oldest branch.

Besides, the execution of instructions along the wrong path may pollute the execution units with useless instructions. As more instructions are dispatched, because multiple paths are active at the same time, the scheduling is more complicated. An example is how to decide between two instructions which have all operands available but only one functional unit is free? If the instruction issued pertains to the wrong path then the resource is wasted with a useless instruction.

Basically, there are two problems with the eager model. First, the large bandwidth necessary to fetch and execute multiple paths and instructions. Second, the pollution generated by useless instructions.

5.4 Reducing Necessary Fetch Bandwidth

Previous works already analyzed the behavior of conditional branches and locality principles which are being used in every single implementation nowadays.

Experiments conduced measured the distance between branches and their taken targets. If the branch is not taken the target is the next instruction (distance=1). If the branch is taken then the target may be inside the same line (IN) or not (OUT). The experiments considered the number of instructions in the cache line (4, 8, 16, 32 and 64), the distance between the branch and the target, the outcome of the branch and the direction (forward/backward). Afterwards, the misprediction rate was measured accordingly with the branch class which they were associated.

It was interesting to see that a considerable amount of mispredictions occurred in branches with very near targets. Starting from 4 instructions per line (16 bytes cache line) the experiments showed that for integer benchmarks (SPECint95) around 4% of all conditional branches have targets inside the same line and 3% of mispredictions occurred when predicting these branches.

With a cache line containing 32 instructions (128 bytes) was reported an average of 54.5% of conditional branches with targets within the same line and almost 60% of mispredictions were concentrated within this subset.

The idea is to take advantage of the spatial locality towards the implementation of multipath. The key is to fetch a long cache line in a single access and then split it into different paths detecting branches and targets within that context.

Considering the processor tends to be much faster than the memory, the split process can be done in the cycle following the fetch if the next cache line to be fetched is known. A cache line predictor may be used instead of a branch predictor. That would be more efficient to allow the split stage to partition the previous line while the fetch is already fetching the next one.

Multiples paths can be delivered to the execution core using only one cache port so reducing the fetch bandwidth necessary. However, the execution bandwidth required is still large, considering that an enormous number of instructions can be issued.

5.5 Reducing Pollution

When multiple paths are pursued, most likely many instructions are replicated. That is, because of conditional branches always merge into a convergent point, many instances of convergent points are activated from each active path.

However, a replicated path is committed only once. All other instances of instructions belonging to a replicated path are canceled as soon as all previous branches are resolved or canceled. Those copies, which are canceled, are considered pollution, since they consume resources but do not commit.

In order to reduce pollution, the convergent paths need to be detected and activated only once. This concept is very similar to the idea of exploit control independence [ROT99]. Convergent paths are control independent but not "data independent". Each instance of a control independent path, in a multipath architecture, pertain to a different data dependency chain.

In this approach, only one instance of a control independent path is activated. This can reduce drastically the number of in-flight instructions reducing pollution and execution bandwidth.

5.6 Replaying Instructions

A control independent path may have instructions which are dependent on results produced by multiple paths pursued previously. To guarantee data dependencies, a control independent path should wait until all previous conditional branches are resolved or canceled.

However, this would delay the execution of such instructions hiding possible benefits. Instead, data instructions which carry true data dependencies and pertain to control independent paths can be issued based on value prediction. If the prediction is not correct the instructions are replayed.

5.7 An Overview of the DCE Architecture

The Dynamic Conditional Execution model is based on the concept that multiple paths are extracted from a single cache line in a single access and sent to the execution core. In other words, execution is done conditionally and instructions are committed according with the dynamic behavior of conditional branches.

Multiple paths are pursued based on the locality of each path. If both paths of a given branch are present they are executed conditionally. Future steps will consider the interaction between the architecture and the compiler.

Merge points are detected and only one path is activated from these points. Input operands are predicted and instructions belonging to control independent paths are issued based on the availability of each operand or a prediction. If a wrong prediction is made, the instruction is replayed.

The fetch is driven by the next cache line prediction [CAL95]. This way, the next line can be fetched while the previous line is segmented into different paths. The next line is supposed to bring more paths to complete the execution of previous conditional branches.

If a line contains a far branch, a prediction is made for the far branch and the next line to be fetched is determined by the branch predictor. The branch predictor holds information about far branches only, minimizing the predictor itself.

Instructions are renamed according with the position of the path on the dynamic tree and value prediction may be used to predict the input operands of instructions belonging to control independent paths which are waiting for such operands.

5.8 Expected Results

DCE is intended to reduce the fetch and execution bandwidths necessary to implement a multipath architecture. Even though DCE is not an eager multipath in its essence, the possibility to execute eagerly both paths of short branches is an attractive approach.

The benefits reside in the fact that it is not necessary to predict short branches. Since they have both targets within the line, they can be executed in a multipath form. Other significant portion of branches have targets in the next sequential line. If the fetch brings sequential lines, those branches will also benefit automatically of a multipath approach.

Minimizing the needs for prediction can improve the prediction itself and the implementation costs. Moreover, the use of multipath for the execution of short branches can isolate up to 60% of mispredictions for a 32 instructions cache line. As the remaining mispredictions are concentrated in easily predictable branches, the performance expected is close to a full eager multipath.

Previous selective multipath, besides the normal predictor, needed also a confidence mechanism to be used in conjunction with the predictor in order to determine whether a branch should be executed eagerly or not.

6. Revisiting Branches

This chapter presents a new insight into branches. Several different cache configurations were simulated in order to study the distribution, target position and behavior of conditional and unconditional branches. Following sections present several analysis which allow to understand better the relations between branches and mispredictions.

In the graphs presented in this chapter, unconditional branches with register based relative targets were not considered. Only direct relative conditional and unconditional branches were measured.

6.1 Branch Direction and Outcome

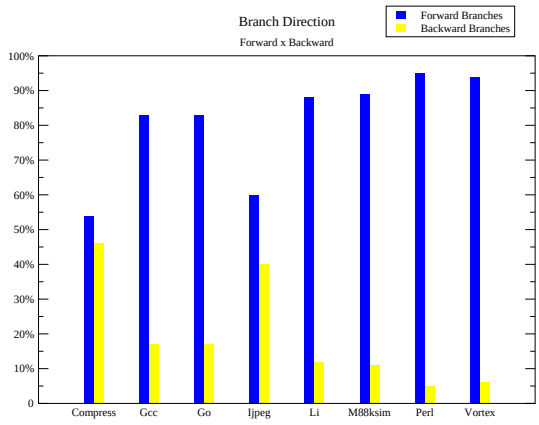

Figure 6.1: Branch Direction

Direction is based on the position of the taken target of a branch instruction. If the taken target is located in an address greater than the address of the branch itself than the target is in a forward position hence the branch is called forward branch. The opposite happens when the target is located before the branch in a smaller address. In this case, the branch is called backward branch.

Figure 6.1 shows the distribution of forward and backward branches for the integer benchmarks of the SPEC95 suite. The direction considered only the position of the taken target and not the outcome of the branch. About 72% of branches have targets in a forward direction but this does not mean that those branches were taken or not.

Figure 6.2 presents the percentage of taken and not taken branches. Here, only the outcome is considered. A taken branch may have a backward or forward target but a not taken branch has always a forward target. About 64% of branches were taken and 36% were not taken.

6.2 Mispredictions based on Direction and Outcome

Figure 6.3 presents mispredictions based on direction. The graph shows the percentage of mispredictions that occurred in forward branches independent of its outcome (taken or not taken). In average, 80% of mispredictions are concentrated in forward branches. Here

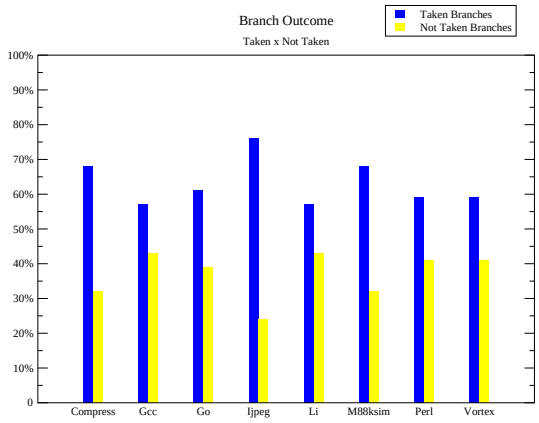

Figure 6.2: Branch Outcome

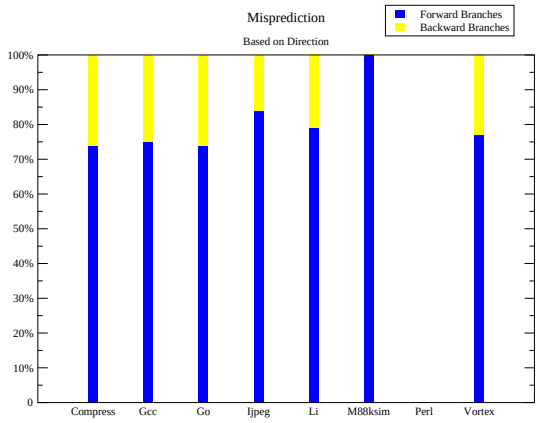

Figure 6.3: Mispredictions based on Direction

again, the direction is based on the position of the target and does not depend on the outcome of the branch nor the size of the cache line.

For benchmark *Perl*, all mispredictions happened for indirect jumps which are not considered here. So, the benchmark presented no mispredictions for direct conditional and unconditional branches.

Figure 6.4 presents mispredictions distributed along with the outcome of the branches. Around 55% of all mispredictions occurred for taken branches. That means that those branches were considered not taken when the prediction was made. On the other hand, 45% of mispredictions happened for not taken branches. In this case, those branches were considered taken when the prediction was made.

Benchmark *Perl* again presented no mispredictions because all mispredictions occurred on indirect branches which were not considered. Although, benchmark *M88ksim* concentrated all mispredictions in taken branches. That means that 100% of mispredictions determined that branches were not taken where they should be assumed taken.

It is possible to perceive that mispredictions are more likely to occur in forward branches than in backward branches. Regarding the outcome, we can see more or less a well distributed occurrence which so far tells that the outcome exercises no strong influence on the prediction, except for benchmark *M88ksim*.

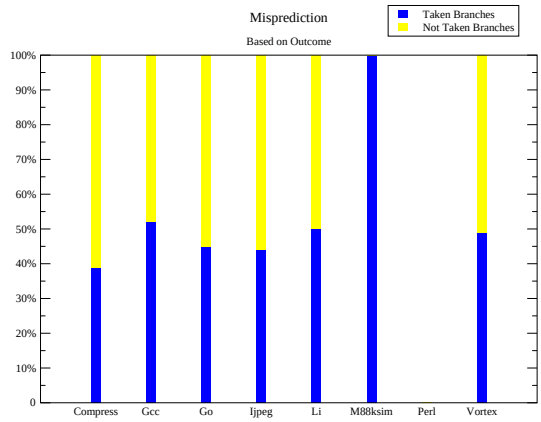

Figure 6.4: Mispredictions based on Outcome

6.3 Target Position - Short Branches

Another analysis is presented considering the position (in/out) of the taken target of direct relative conditional and unconditional branches regarding a certain distance.

Again, the taken target is considered when classifying a branch as in or out. The taken target is used because that is the farthest target possible of a branch. The not taken target is always the next instruction. In this analysis, a branch which the taken target is within a certain distance is called short branch for the distance considered.

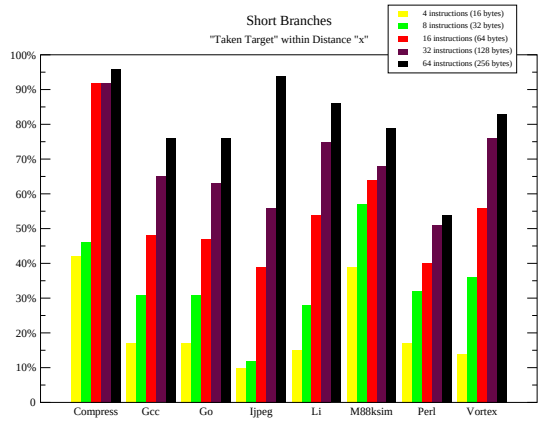

Figure 6.5: Short Branches for Different Distances

Figure 6.5 shows the position of the taken targets of all direct conditional and unconditional branches according with a certain distance. The averages are shown on table 6.1. For a 4 instructions distance (16 bytes), the average number of short branches is 22%. The average raises to 34.43% of short branches when the distance is increased to 8 instructions or 32 bytes. For a 16 instructions distance the average is 57.14%, for 32 instructions is 70.71% and for a 64 instructions distance the average is 84.29% of short branches.

The distance is measured from the branch to the taken target. If the distance of the taken target is within the distance considered then the branch is called short branch.

Table 6.1: Short Branches and Mispredictions

insn/line (bytes)	% of short branches average	% of Mispredictions within short branches
4 (16)	22.00	36.57
8 (32)	34.43	48.43
16 (64)	57.14	65.86
32 (128)	70.71	83.57
64 (256)	84.29	93.00

The analysis is based on the taken target disregarding the outcome of the branch because the not taken target is always the next sequential instruction. So the distance between the branch and its not taken target is always 1 instruction.

Usually, the worst case is when the branch is taken because that requires a second access to the cache to bring the taken target if it is not in the same line. Furthermore, the instructions that are between the branch and the taken target need to be discarded even though they can be already into the fetch unit if the when branch is predicted taken.

As the worst case is the taken branches we conduced this experiment looking to the position of the taken targets of every direct branch/jump. The outcome was not important at this time because we wanted to compute the average number of branches with the farthest target within a distance. So the taken target give us the maximum distance independent of the outcome of the branch.

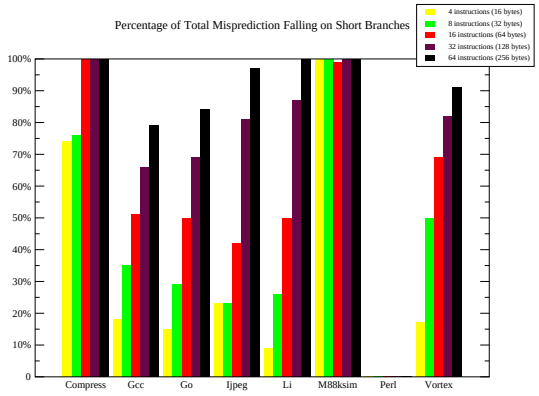

Figure 6.6: Mispredictions Falling on Short Branches

Figure 6.6 presents the percentage of mispredictions that felt within short branches. The averages are presented on table 6.1, third column. For a 4 instructions distance, 36.57% of all mispredictions occurred into the 22% of short branches, and so on. For a 8 instructions distance, the short branches concentrated an average of 48.43% of all mispredictions. 65.86%, 83.57% and 93% of mispredictions occurred in short branches for the 16, 32 and 64 instructions distances, respectively.

Benchmarks *M88ksim* and *Perl* presented interesting results. For *M88ksim* all mispredictions occurred within short branches starting from 4 instructions distances. For *Perl*, no misprediction at all was reported within short branches from 4 to 64 instructions line. This means that mispredictions occurred only for indirect branches or for branches with taken targets more than 64 instructions far from the branch itself. *Perl* was not considered on the averages presented on this section.

7. Microarchitecture Overview

DCE is being proposed as a new architecture model to reduce misprediction penalties. By applying multipath techniques to short branches and predicting the behavior and outcome of far branches, DCE is expected to produce good results based on the assumptions below.

We define initially, a short branch as a branch which has the taken target within a certain distance from the branch itself. A far branch is considered to be a branch which targets a position outside of that distance.

The intention of DCE is to reduce short branch mispredictions as close to zero as possible. The oracle DCE would eliminate completely any misprediction within short branches.

Table 7.1 presents a comparison of the performance of an Oracle DCE machine against the performance of a base and an ideal machines. The base machine is a 16 way superscalar with two-level g-share branch prediction and perfect cache system (i.e., no misses). The ideal machine is the base machine with perfect branch prediction.

There are two Oracle DCEs machines. The first one uses perfect prediction for only short forward branches. The second one, uses perfect prediction for every short branch, forward or backward. Both DCE machines considered short branches those branches with taken targets within 32 instructions from the branch. An open question is to determine the ideal distance for classifying short branches.

The effect expected is to get the upper bound performance of a DCE machine when multipath is applied to short branches. When multipath is applied to a branch, the effect is the same as predict it with 100% accuracy.

However, as perfect prediction is applied, only the correct path is executed so the resource conflicts are not modeled in the experiments reported below. Also, data is always dependent on the correct path. As a result, the performance of a real DCE machine is expected to be below the performance indexes showed by the oracle machines since executing both paths of a branch will demand more resources and cause some instructions to be re-executed. Therefore, an optimistic upper-bound is shown.

Table 7.1: Potential Upper Bound Performance of DCE

Benchmark	Base			Oracle DCE FW IN			Oracle DCE FW BW IN			Ideal Perfect Prediction		
	Speed-up	IPC	Gap %	Speed-up	IPC	Gap %	Speed-up	IPC	Gap %	Speed-up	IPC	Gap %
Compress	1	1.321	80.79	3.21	4.238	38.38	3.21	4.238	38.38	5.21	6.877	0
Gcc	1	2.196	68.07	1.93	4.238	38.37	1.93	4.238	38.38	3.13	6.877	0
Go	1	1.777	70.58	1.49	2.651	56.11	1.59	2.82	53.26	3.40	6.040	0
Ijpeg	1	4.219	49.99	1.50	6.330	24.98	1.58	6.682	20.8	2.0	8.438	0
Average	1	2.38	67.36	2.03	4.36	39.46	2.08	4.50	37.7	3.43	7.06	0

We measured IPC (Instructions per Cycle), speed-up and gap. Speed-up is the ratio between the IPC of the improved machine and the IPC of the base machine. Gap is the percentage of Ideal performance unmet by a given architecture, in terms of IPC.

So the speed-up tells how faster is a given architecture over the base architecture and the gap tells how much of the ideal performance is still unmet by a given architecture. In fact, speed-up and gap are redundant metrics used for the only purpose of quantifying the existent room for improvement.

By comparing the performance of both base and ideal performance we intended to show that there is room for improvement. The goal was to show that about 67.36% of the ideal

performance is unmet by the base architecture just because the base architecture relies on branch prediction.

We presented also the performance of two different DCE Oracle machines. The analysis has a twofold goal. First, show that with DCE we can reduce the gap in almost 40% relative to the performance of the Ideal machine. Second, between the two DCE machines we can realize the impact of DCE for both forward and backward branches.

There is not a big difference between the two DCE Oracle machines. This means that applying DCE for only forward branches can result in a good performance or almost all the performance gain can be obtained by just applying DCE for forward branches. However, it is not possible to confirm it at this point which one of the models will deliver the best performance since other factors as resource contention were not simulated.

7.1 Microarchitecture Specification

We strongly believe that concentrating efforts on reducing mispredictions for short branches only is worth. First, because short branches can be easily executed eagerly without requiring the same amount of resources needed to execute all branches eagerly. Second, because most of mispredictions happen on those branches, to concentrate efforts on that sounds very promising.

7.1.1 Fetch

Fetching instructions in DCE will be easy. A predictor will determine what is the next line to be fetched as in a conventional superscalar architecture. However, the predictor only determines a new line when a far branch is fetched.

When determining the distance which will define whether a branch is short or far, we will consider the amount of resources to execute eagerly all branches that will fall in the short branch category. Determining the correct distance may cause a big impact in the implementation but also leaves an interesting way to apply a very flexible implementation.

For example, let's define a short branch as any branch which has the target within 4 instructions from the branch itself. So, if a branch has the target within 5 instructions from it then that branch is deemed a far branch. A far branch will discontinue the multipath execution of previous short branches.

In fact, DCE executes eagerly all short branches until it founds a far branch. A far branch is predicted using a branch or cache line predictor (discussed latter). So, the flexibility resides in the fact that it is possible to determine, even dynamically, what is the distance which is considered for eagerly execution. The more distant the target is, when executing eagerly, the more branches will be executed in a multipath fashion and more short branches probably will appear in a contiguous sequence.

In other words, the distance chosen will determine how many short branches will appear between two far branches. If the defined distance captures all branches, then all branches are executed eagerly and no mispredictions occur at all. Obviously, the distance must be such as to avoid a large number of resources but an acceptable benefit in order to justify the effort. In the same sense, if the distance is 1, all branches will be considered far branches thus all of them will be predicted and the behavior will be the same as a normal superscalar architecture.

The fetch will start from the entry point of a program and will fetch sequentially throughout the program until a far branch is fetched. When a far branch is found, the predictor must recognize the far branch and redirect the fetch to the far target address.

Short branches found in the middle of cache lines are just sent down the pipeline, together with all instructions contained in the line. No selection is made at the fetch or decoder in order to decide whether an instruction has to be sent or not to the pipeline. All instructions contained on the line are sent, except those that come before or after a far branch.

So a small limit distance chosen will determine more far branches therefore more predictions and more predisposition for mispredictions. Larger distances will determine more short branches and more multipath resources will be needed to execute them. However, less mispredictions are likely to happen.

7.1.2 Branch Predictor

Because fetch redirection will be done only when far branches are fetched, the predictor must recognize and thus predict only far branches.

For larger distances, the architecture must be able to execute more short branches in a multipath fashion. On the other hand, the fetch is simplified because less redirections are likely to occur.

The term simplified means smaller branch table since less branches need to be predicted. Also, prefetching a sequential line does not require any kind of information being transferred between the prefetch and the predictor.

There are two ideas regarding the predictor. First is to use a small table and store there only far branches. If a line brought from the icache hits into the branch table that means that a far branch is inside the line. So the prediction is used to determine the next line and this information is bypassed to the following stages in order to proceed with the multipath conclusion for the previous outstanding branches.

The balance between the size of the prediction table and the amount of resources to execute the short branches will be defined by the distance limit used to determine whether a branch is short or far.

Another idea is to use a next cache line predictor as proposed in [CAL95]. Therefore, no branch is handled at the fetch but only the cache line and set. For a given cache line the predictor must tell the fetch unit which is the next cache line and set. The trade-off between these two alternatives will be considered when determining which is the best scheme to be used with DCE.

7.1.3 Decode

Instructions are decoded normally. The major issue on the decode is attributed to the minimum number of instructions decoded per cycle in order to allow a sufficient amount of instructions to be sent to the rest of the pipeline in order to keep an acceptable performance benefit.

An important information regarding branches is to decode the offset for direct conditional and unconditional branches. This information is important to the renaming stage in order to capture the beginning and end of each path.

The decode itself can be done normally without requiring unconventional processes to be added due to the DCE concept implementation.

7.1.4 Renaming

Renaming is probably one of the most important, perhaps complexes, parts of DCE. The efficiency of the renaming will most likely determine the gains of applying the technique.

The reason is because multiple instructions belonging to different paths will be in execution at the same time. It is possible to isolate the effect of each instruction's execution by using multiple register files as used in simultaneous multithreading architectures. But this would cost probably too much to justify the implementation of DCE. It is necessary to have a large number of instructions, being executed independently, to justify the allocation of an entire register file just to compensate some temporary false data dependencies.

As considered paths will be short, because of only short branches are considered for multipath execution, the number of instructions belonging to an unique path will not be very large. Thus allocate an entire register file to separate the execution and facilitate the squash afterwards is probably not the best solution.

Instead, the renaming phase will be responsible for renaming instructions accordingly with the path which they belong and handle all data dependencies regardless of the position of the instruction and whether the instructions will be squashed or not.

The renaming will be done based on the limits of each path (i.e. basic block). The information extracted from the branches during the decode phase (offsets) will be used to determine how to rename the instruction.

The R10000 renaming scheme [MIP95] adapted previously to be used in the Mulflux project [SAN99, CHA99] will be slightly modified to be used in DCE [SAN01].

The renaming implements a free register list and an active register list just as done in the R10000. The free list handles all registers not yet used which are available for renaming. The active list holds all output registers which are currently in use by an in-flight instruction. A register is released (transferred from the active list to the free list) only when no further instruction depends on its current value.

In the Mulflux project, two modifications were added to the original scheme. First, multiple mapping tables were introduced to handle the temporary false data dependencies (discussed latter). Second, an active register's counter was associated to each physical register in order to avoid two entries of the same register into the free list.

Notice that using multiple mapping tables, with active register counters, requires only one physical register file, one free register list and one active register list.

7.1.5 Handling Multiple Mapping Tables

Renaming is done as on the R10000 renaming procedure. Except for the multiple tables and the counters associated to each physical register, the procedure is very similar.

When an instruction is renamed, a physical register id is retrieved from the free register list. The logical register is then mapped to a physical register and the mapping information is stored into the mapping table.

When a short branch is found, a new mapping table is allocated. The new mapping table is a copy of the current mapping table, being used to rename the current path. Therefore, the new mapping table inherits automatically all mappings done previously. A second mapping table can be also allocated/created to handle the renaming for the control independent path.

The new mapping table will be used afterwards when the instructions from the taken path are renamed. The renaming process is done sequentially throughout the fetch buffer, renaming all instructions in the order they were fetched.

Instructions that come right after a conditional branch are instructions that belong to the not-taken path. Those instructions are renamed using the current mapping associations, present into the current mapping table. Notice that the new table allocated is not used at this time nor the second table which will be used latter on.

When the not-taken path finishes, the target instruction of the last branch is reached at the renaming, the renaming starts to use the mappings stored into the new mapping table. Recall that the new mapping table holds exactly the mappings as of the time the branch was renamed and is updated with the mappings corresponding to the taken path instructions. So, the new mapping table will hold the mappings to the taken path and the previous mapping table holds the mappings for the not-taken.

All data dependencies are respected with this strategy and a recovery is easily done by just releasing the register stored in the mapping table which holds the mappings for the wrong-path.

The counters are used to guarantee that only one entry of a given register is inserted into the free list by the time the register is released. Because of the nature of the DCE model, mapping tables may have identical mappings at one time but they may have two different instructions writing at the same output register (temporary output dependence).

If an instruction writes into a previously mapped register, the previous register must go to the active list. But, that may happen also for another table when renaming an instruction for another path. In this case, the same register is inserted into the active list twice. Each time a register is inserted into the active list its counter is incremented, so it reflects the number of times a register was written in different paths of execution.

A register is moved from the active list to the free list only if its counter is zero. Thus guaranteeing that there is only one entry into the free list [CHA99].

7.1.6 Control and Data Independence

Despite its ability to execute multiple paths of certain branches, DCE is also intended to take advantage of control independence [ROT99b].

Accordingly with the previous description of the renaming process, two tables can be created after a short branch which is going to be executed eagerly. Both tables are copies of the current table and one will be used to rename the instructions on the taken path and the other one will be used to rename the instruction on the control independent path.

A second table is used to rename the instructions on the control independent path so the dependencies are easier handled by the instructions that are coming. This because a new branch may fork from the control independent path and then the dependencies are correctly kept.

When the control independent point is reached, the table created previously also holds the mappings as of the time the corresponding branch was renamed.

Notice that at the time the CI code is been renamed there are two tables. One holds the mappings for the main path and the not-taken path and the other one holds the main path mappings and the taken path mappings.

When an instruction on the CI path is renamed and its input operands were renamed only once (i.e. same mappings in both tables) that means that no writing was done on the register during the taken or not-taken path but only before the branch. So, a instruction which has the same mappings on both tables, for the input operands, or have all immediate operands is also data independent in this context.

A control and data independent instruction (CIDI) can be executed as soon as all inputs operands are ready and it will not be squashed if any previous far branch was correctly predicted.

In DCE, far branches are predicted as usual. Thus, if a far branch is mispredicted a full squash is done. The pipeline is flushed and all mapping tables are released. If all previous far branches were correctly predicted any short branch previously executed will not result in misprediction, because both paths are executed, so CIDI instructions can be committed.

If an instruction has two different mappings on the taken and not-taken tables, is because the instruction was written during the conditional execution of one of the paths. In this case, does not matter which path wrote into the register, but the instruction would have to be stalled until the branch and the input operands are ready.

To avoid stall an instruction and delay its issue, DCE will issue CIDD (control independent data dependent) instruction based on value prediction. Therefore all instructions belonging to the CI path can be issued. CIDI instruction are issued normally and CIDD instructions are issued upon o value prediction.

The value prediction will allow to issue the CIDD instruction regardless of the outcome of the previous branch and even when the input operands are not ready. Thus, this will accelerate the execution of the CI path. If the value predicted is wrong the instruction is re-executed latter.

Re-execution will require a selective re-issue of CIDD instructions with mispredicted values. The re-issuing technique will be part of the future work. Another alternative is stall the CIDD instruction until the branch is resolved and the correct input register has been written.

7.1.7 Scheduling

After renamed, an instruction is tagged with the mapping table *id* used to rename it. The mapping table *id* is also used latter to determine the beginning and the end of a given basic block by the time of retirement.

An instruction already renamed, which has all input operands ready, can be issued normally. However, the scheduler has to stall CIDD instructions if no value prediction is used. Or, if value prediction is used, the scheduler has to wait until the predictor can inform what is the predicted input value.

The scheduler is not supposed to apply any priority in order to schedule an instruction from the taken, not-taken or CI paths since no prediction of any kind is used for this purpose. However, depending on the number of instructions and outstanding branches this may be something to consider in order to allocate more efficiently the resources available.

7.1.8 Retiring

Instructions are fetched, decoded and renamed in order. The dispatch (scheduling and issue) is where instructions are placed out of its original order in order to overpass data dependencies.

A reorder buffer will be used to keep the original order. An instruction can be retired if it reaches the top of the reorder buffer.

When a branch reaches the top of the reorder buffer, all previous instructions are already committed and the architectural state is deemed correct. If the branch was predicted (far branch), then the prediction is compared against the correct condition. If the prediction is incorrect, all instructions are flushed and the fetch is restarted from the correct target address.

If the branch is correctly predicted, nothing occurs and the retiring continues since the instructions that were fetched after the branch are in the correct path.

If the branch is a short branch, no prediction was made and both not-taken and taken paths are contiguously present into the reorder buffer. When the branch reaches the top of the reorder buffer, and is ready to commit, the condition is known and the decision of which path is correct can be taken at this time.

If the branch is taken, the instructions that follow the branch into the reorder buffer must be canceled until the first instruction of the taken path reaches the top of the reorder buffer. The mapping table *id*, is used to detect the end of the not-taken path.

When a different tag is found, that indicates that the beginning of the taken path was reached. If there were nested short branches, the same procedure applies for the renaming, detection, commit and squash of instruction from the correct and wrong paths.

By checking the tag of an instruction it is possible to keep track of what mapping table was used and easily keep a dynamic hierarchy associating paths to branches and vice-versa.

7.2 Final Considerations

Once more, DCE has been thought as a simple form of multipath. We strongly believe that DCE will provide great results. Our primary goal is to address mispredictions on short branches, since those can be executed eagerly without putting excessive pressure on the fetch unit nor on the execution core.

Indeed, a more complex renaming and a recovery scheme to handle data misprediction is required. On the other hand, the cost of the multiple tables for renaming and the data prediction tables will be partially absorbed by reducing the size of the branch tables since only far branches need to be memorized.

An open question is to understand exactly how the architecture can interact with the compiler. We know that the compiler can optimize the source code in order to reduce the number of branches, invert the direction or even the outcome of most of the branches by applying optimization techniques. However, we know that mispredictions still occur even for the most optimized codes.

A comparison against other schemes is necessary. It is important to show that DCE is relatively simple when compared with other alternatives such as trace cache [ROT96, RAM00], trace processors [SMI97], dynamic predication [KLA98b] and other multiple branch predictor techniques [MIC99, YEH93b, RAK00] as well as other multipath architectures [SKA99, ?, KLA98a].

8. Appendix A

Table 8.1: Branch Profile for a 4 Instructions Distance (16 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.539	0.768	0.809	0.568	0.774	0.750	0.905	0.849	0.671
BW	0.461	0.232	0.191	0.432	0.226	0.250	0.095	0.151	0.329
IN	0.419	0.170	0.167	0.104	0.149	0.392	0.165	0.140	0.215
OUT	0.581	0.830	0.833	0.896	0.851	0.608	0.835	0.860	0.785
Taken	0.680	0.570	0.607	0.765	0.579	0.679	0.591	0.593	0.655
NOT Taken	0.320	0.430	0.393	0.235	0.421	0.321	0.409	0.407	0.345
FW IN Taken	0.280	0.095	0.111	0.063	0.105	0.214	0.116	0.094	0.137
FW IN NOT Taken	0.139	0.055	0.054	0.018	0.033	0.107	0.050	0.043	0.066
BW IN Taken	0.000	0.011	0.002	0.014	0.005	0.071	0.000	0.003	0.007
BW IN NOT Taken	0.000	0.009	0.001	0.009	0.006	0.000	0.000	0.001	0.005
FW OUT Taken	0.120	0.297	0.357	0.315	0.310	0.215	0.409	0.355	0.272
FW OUT NOT Taken	0.000	0.321	0.287	0.172	0.326	0.214	0.331	0.358	0.195
BW OUT Taken	0.280	0.167	0.137	0.372	0.159	0.179	0.066	0.141	0.239
BW OUT NOT Taken	0.181	0.046	0.051	0.036	0.055	0.000	0.029	0.006	0.079

Table 8.2: Misprediction Profile for a 4 Instructions Distance (16 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.743	0.763	0.736	0.856	0.735	1.000	0.000	0.708	0.738
BW	0.257	0.237	0.264	0.145	0.265	0.000	0.000	0.292	0.262
IN	0.743	0.178	0.149	0.230	0.086	0.996	0.000	0.168	0.309
OUT	0.257	0.822	0.851	0.771	0.914	0.004	0.000	0.837	0.692
Taken	0.427	0.542	0.455	0.444	0.518	1.000	0.000	0.510	0.483
NOT Taken	0.573	0.458	0.545	0.556	0.482	0.000	0.000	0.490	0.517
FW IN Taken	0.316	0.085	0.074	0.068	0.040	0.996	0.000	0.089	0.141
FW IN NOT Taken	0.427	0.057	0.070	0.067	0.033	0.000	0.000	0.052	0.151
BW IN Taken	0.000	0.016	0.002	0.034	0.004	0.000	0.000	0.004	0.005
BW IN NOT Taken	0.000	0.021	0.003	0.060	0.010	0.000	0.000	0.022	0.011
FW OUT Taken	0.000	0.347	0.291	0.330	0.390	0.004	0.000	0.307	0.236
FW OUT NOT Taken	0.000	0.274	0.300	0.389	0.273	0.000	0.000	0.265	0.210
BW OUT Taken	0.111	0.095	0.087	0.013	0.083	0.000	0.000	0.110	0.101
BW OUT NOT Taken	0.146	0.107	0.172	0.039	0.167	0.000	0.000	0.156	0.145

Table 8.3: Branch Profile for a 8 Instructions Distance (32 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.539	0.768	0.809	0.568	0.774	0.750	0.905	0.849	0.671
BW	0.461	0.232	0.191	0.432	0.226	0.250	0.095	0.151	0.329
IN	0.459	0.311	0.315	0.121	0.283	0.570	0.322	0.357	0.301
OUT	0.541	0.689	0.685	0.879	0.717	0.430	0.678	0.643	0.699
Taken	0.680	0.570	0.607	0.765	0.579	0.679	0.591	0.593	0.655
NOT Taken	0.320	0.430	0.393	0.235	0.421	0.321	0.409	0.407	0.345
FW IN Taken	0.320	0.155	0.175	0.063	0.136	0.285	0.157	0.250	0.178
FW IN NOT Taken	0.139	0.112	0.101	0.021	0.085	0.214	0.165	0.092	0.093
BW IN Taken	0.000	0.026	0.027	0.014	0.026	0.071	0.000	0.012	0.017
BW IN NOT Taken	0.000	0.018	0.011	0.023	0.035	0.000	0.000	0.003	0.013
FW OUT Taken	0.080	0.237	0.293	0.315	0.278	0.143	0.368	0.199	0.231
FW OUT NOT Taken	0.000	0.263	0.240	0.169	0.275	0.108	0.215	0.309	0.168
BW OUT Taken	0.280	0.152	0.112	0.372	0.138	0.179	0.066	0.133	0.229
BW OUT NOT Taken	0.181	0.036	0.040	0.022	0.026	0.000	0.029	0.003	0.070

Table 8.4: Misprediction Profile for a 8 Instructions Distance (32 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.763	0.759	0.735	0.847	0.767	1.000	0.000	0.733	0.747
BW	0.237	0.241	0.265	0.153	0.232	0.000	0.000	0.271	0.254
IN	0.763	0.351	0.293	0.228	0.256	0.996	0.000	0.495	0.475
OUT	0.237	0.649	0.707	0.772	0.744	0.004	0.000	0.505	0.525
Taken	0.415	0.537	0.453	0.442	0.498	1.000	0.000	0.520	0.481
NOT Taken	0.585	0.463	0.547	0.558	0.502	0.000	0.000	0.484	0.520
FW IN Taken	0.322	0.156	0.123	0.070	0.038	0.996	0.000	0.247	0.212
FW IN NOT Taken	0.441	0.115	0.111	0.066	0.032	0.000	0.000	0.116	0.196
BW IN Taken	0.000	0.033	0.018	0.031	0.060	0.000	0.000	0.034	0.021
BW IN NOT Taken	0.000	0.048	0.042	0.063	0.125	0.000	0.000	0.098	0.047
FW OUT Taken	0.000	0.272	0.246	0.325	0.369	0.004	0.000	0.200	0.179
FW OUT NOT Taken	0.000	0.215	0.256	0.387	0.329	0.000	0.000	0.171	0.161
BW OUT Taken	0.093	0.075	0.067	0.017	0.031	0.000	0.000	0.040	0.069
BW OUT NOT Taken	0.145	0.086	0.138	0.042	0.015	0.000	0.000	0.098	0.117

Table 8.5: Branch Profile for a 16 Instructions Distance (64 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.539	0.768	0.809	0.568	0.774	0.750	0.905	0.849	0.671
BW	0.461	0.232	0.191	0.432	0.226	0.250	0.095	0.151	0.329
IN	0.920	0.477	0.474	0.387	0.540	0.642	0.397	0.560	0.565
OUT	0.080	0.523	0.526	0.613	0.460	0.358	0.603	0.440	0.435
Taken	0.680	0.570	0.607	0.765	0.579	0.679	0.591	0.593	0.655
NOT Taken	0.320	0.430	0.393	0.235	0.421	0.321	0.409	0.407	0.345
FW IN Taken	0.320	0.228	0.242	0.231	0.223	0.357	0.227	0.296	0.255
FW IN NOT Taken	0.139	0.169	0.164	0.119	0.194	0.214	0.165	0.208	0.148
BW IN Taken	0.280	0.052	0.047	0.014	0.065	0.071	0.000	0.050	0.098
BW IN NOT Taken	0.181	0.028	0.022	0.024	0.058	0.000	0.004	0.005	0.064
FW OUT Taken	0.080	0.164	0.226	0.148	0.192	0.072	0.298	0.152	0.154
FW OUT NOT Taken	0.000	0.206	0.178	0.070	0.165	0.107	0.215	0.193	0.114
BW OUT Taken	0.000	0.126	0.093	0.372	0.100	0.178	0.066	0.094	0.148
BW OUT NOT Taken	0.000	0.026	0.030	0.022	0.003	0.000	0.025	0.001	0.020

Table 8.6: Misprediction Profile for a 16 Instructions Distance (64 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.743	0.745	0.735	0.834	0.779	1.000	0.000	0.725	0.737
BW	0.257	0.256	0.265	0.166	0.221	0.000	0.000	0.275	0.263
IN	1.000	0.507	0.498	0.416	0.503	0.992	0.000	0.686	0.673
OUT	0.000	0.494	0.502	0.584	0.497	0.008	0.000	0.319	0.329
Taken	0.389	0.528	0.451	0.428	0.524	0.992	0.000	0.505	0.468
NOT Taken	0.611	0.472	0.549	0.571	0.476	0.008	0.000	0.499	0.533
FW IN Taken	0.307	0.218	0.203	0.163	0.187	0.984	0.000	0.281	0.252
FW IN NOT Taken	0.436	0.163	0.190	0.162	0.111	0.008	0.000	0.155	0.236
BW IN Taken	0.081	0.043	0.033	0.029	0.096	0.000	0.000	0.045	0.051
BW IN NOT Taken	0.176	0.082	0.072	0.061	0.109	0.000	0.000	0.200	0.133
FW OUT Taken	0.000	0.200	0.168	0.220	0.230	0.008	0.000	0.154	0.130
FW OUT NOT Taken	0.000	0.164	0.175	0.288	0.251	0.000	0.000	0.135	0.118
BW OUT Taken	0.000	0.067	0.047	0.016	0.010	0.000	0.000	0.022	0.034
BW OUT NOT Taken	0.000	0.063	0.112	0.060	0.006	0.000	0.000	0.008	0.046

Table 8.7: Branch Profile for a 32 Instructions Distance (128 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.539	0.768	0.809	0.568	0.774	0.750	0.905	0.849	0.671
BW	0.461	0.232	0.191	0.432	0.226	0.250	0.095	0.151	0.329
IN	0.920	0.647	0.633	0.559	0.752	0.679	0.508	0.756	0.690
OUT	0.080	0.353	0.367	0.441	0.248	0.321	0.492	0.244	0.310
Taken	0.680	0.570	0.607	0.765	0.579	0.679	0.591	0.593	0.655
NOT Taken	0.320	0.430	0.393	0.235	0.421	0.321	0.409	0.407	0.345
FW IN Taken	0.320	0.281	0.317	0.288	0.264	0.357	0.293	0.361	0.302
FW IN NOT Taken	0.139	0.233	0.229	0.173	0.338	0.214	0.207	0.339	0.194
BW IN Taken	0.280	0.091	0.060	0.072	0.091	0.107	0.004	0.050	0.126
BW IN NOT Taken	0.181	0.042	0.026	0.025	0.059	0.000	0.004	0.006	0.069
FW OUT Taken	0.080	0.111	0.150	0.091	0.151	0.072	0.231	0.087	0.108
FW OUT NOT Taken	0.000	0.142	0.113	0.016	0.021	0.107	0.174	0.062	0.068
BW OUT Taken	0.000	0.087	0.079	0.314	0.073	0.143	0.062	0.094	0.120
BW OUT NOT Taken	0.000	0.013	0.025	0.020	0.003	0.000	0.025	0.001	0.015

Table 8.8: Misprediction Profile for a 32 Instructions Distance (128 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.738	0.745	0.735	0.837	0.787	1.000	0.000	0.737	0.739
BW	0.263	0.254	0.264	0.163	0.213	0.000	0.000	0.263	0.261
IN	1.000	0.662	0.693	0.814	0.871	0.996	0.000	0.820	0.794
OUT	0.000	0.338	0.307	0.186	0.130	0.004	0.000	0.180	0.206
Taken	0.374	0.530	0.449	0.427	0.511	0.996	0.000	0.472	0.456
NOT Taken	0.626	0.471	0.550	0.573	0.489	0.004	0.000	0.528	0.544
FW IN Taken	0.305	0.275	0.284	0.347	0.361	0.992	0.000	0.329	0.298
FW IN NOT Taken	0.433	0.215	0.278	0.375	0.310	0.004	0.000	0.255	0.295
BW IN Taken	0.070	0.062	0.042	0.029	0.088	0.000	0.000	0.047	0.055
BW IN NOT Taken	0.193	0.110	0.089	0.063	0.111	0.000	0.000	0.193	0.146
FW OUT Taken	0.000	0.146	0.087	0.039	0.050	0.004	0.000	0.080	0.078
FW OUT NOT Taken	0.000	0.109	0.087	0.076	0.065	0.000	0.000	0.073	0.067
BW OUT Taken	0.000	0.047	0.038	0.011	0.011	0.000	0.000	0.019	0.026
BW OUT NOT Taken	0.000	0.036	0.096	0.059	0.003	0.000	0.000	0.007	0.035

Table 8.9: Branch Profile for a 64 Instructions Distance (512 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.539	0.768	0.809	0.568	0.774	0.750	0.905	0.849	0.671
BW	0.461	0.232	0.191	0.432	0.226	0.250	0.095	0.151	0.329
IN	0.960	0.760	0.765	0.940	0.862	0.786	0.537	0.825	0.856
OUT	0.040	0.240	0.235	0.060	0.138	0.214	0.463	0.175	0.144
Taken	0.680	0.570	0.607	0.765	0.579	0.679	0.591	0.593	0.655
NOT Taken	0.320	0.430	0.393	0.235	0.421	0.321	0.409	0.407	0.345
FW IN Taken	0.360	0.320	0.361	0.379	0.302	0.358	0.310	0.390	0.355
FW IN NOT Taken	0.139	0.285	0.270	0.188	0.356	0.321	0.219	0.379	0.220
BW IN Taken	0.280	0.107	0.095	0.333	0.142	0.107	0.004	0.050	0.204
BW IN NOT Taken	0.181	0.048	0.039	0.041	0.062	0.000	0.004	0.006	0.077
FW OUT Taken	0.040	0.072	0.107	0.000	0.113	0.071	0.215	0.058	0.055
FW OUT NOT Taken	0.000	0.090	0.072	0.002	0.003	0.000	0.161	0.022	0.041
BW OUT Taken	0.000	0.071	0.044	0.053	0.022	0.142	0.062	0.094	0.042
BW OUT NOT Taken	0.000	0.006	0.013	0.005	0.000	0.000	0.025	0.001	0.006

Table 8.10: Misprediction Profile for a 64 Instructions Distance (512 bytes)

Benchmark	Compress	Gcc	Go	Ijpeg	Li	M88ksim	Perl	Vortex	Avg
FW	0.734	0.743	0.736	0.841	0.807	1.000	0.000	0.758	0.743
BW	0.266	0.257	0.264	0.159	0.193	0.000	0.000	0.245	0.258
IN	1.000	0.794	0.838	0.973	0.996	1.000	0.000	0.910	0.886
OUT	0.000	0.206	0.162	0.027	0.004	0.000	0.000	0.089	0.114
Taken	0.374	0.530	0.448	0.432	0.525	0.992	0.000	0.490	0.460
NOT Taken	0.626	0.470	0.552	0.568	0.475	0.008	0.000	0.510	0.540
FW IN Taken	0.303	0.329	0.317	0.392	0.437	0.992	0.000	0.387	0.334
FW IN NOT Taken	0.431	0.257	0.320	0.449	0.365	0.008	0.000	0.304	0.328
BW IN Taken	0.071	0.082	0.060	0.029	0.085	0.000	0.000	0.045	0.065
BW IN NOT Taken	0.195	0.126	0.140	0.103	0.107	0.000	0.000	0.178	0.160
FW OUT Taken	0.000	0.092	0.052	0.000	0.001	0.000	0.000	0.040	0.046
FW OUT NOT Taken	0.000	0.067	0.046	0.000	0.003	0.000	0.000	0.028	0.035
BW OUT Taken	0.000	0.027	0.018	0.010	0.000	0.000	0.000	0.018	0.016
BW OUT NOT Taken	0.000	0.020	0.046	0.016	0.000	0.000	0.000	0.003	0.017

9. Appendix B

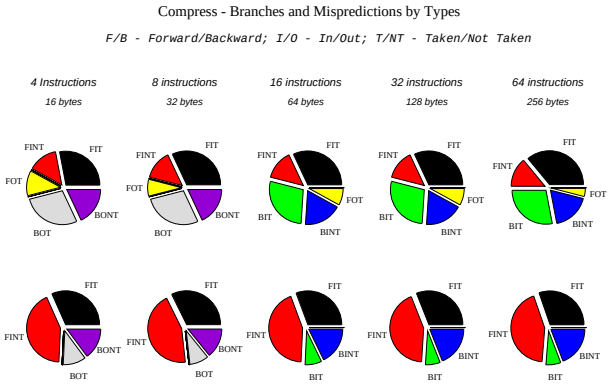

Figure 9.1: Short branches and Mispredictions - Compress

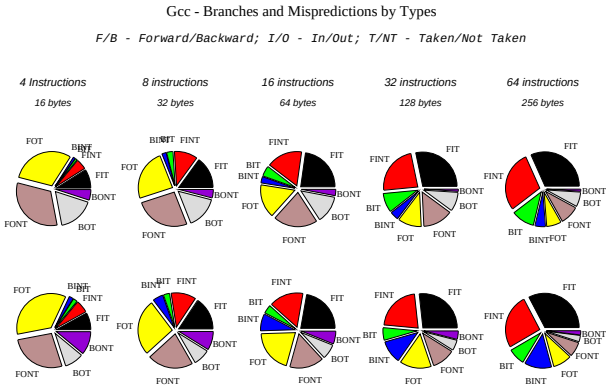

Figure 9.2: Short branches and Mispredictions - Gcc

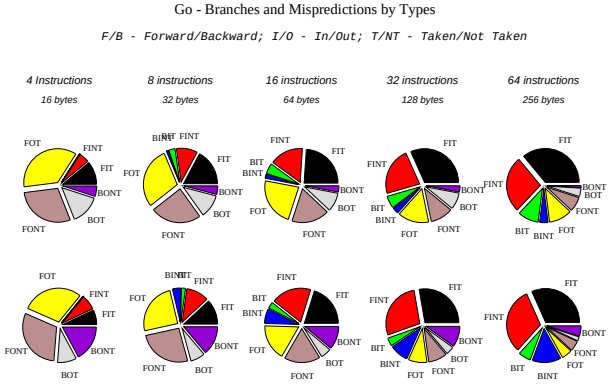

Figure 9.3: Short branches and Mispredictions - Go

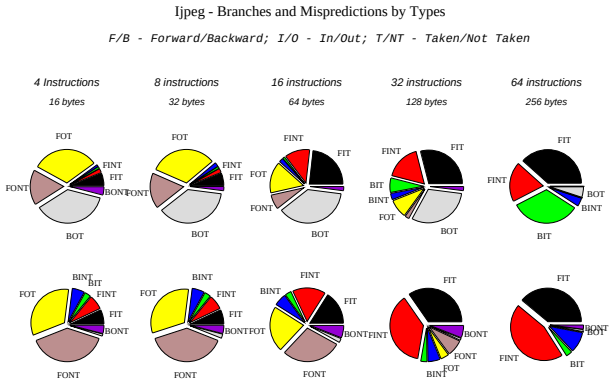

Figure 9.4: Short branches and Mispredictions - Ijpeg

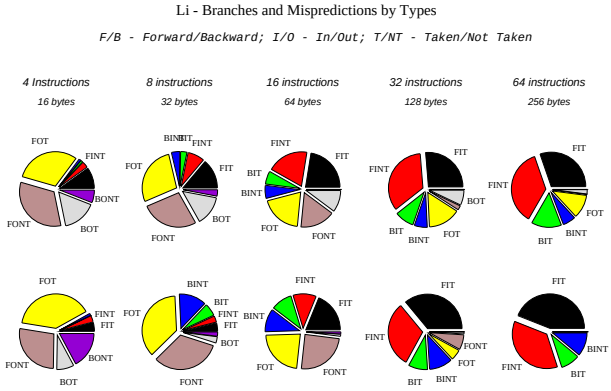

Figure 9.5: Short branches and Mispredictions - Li

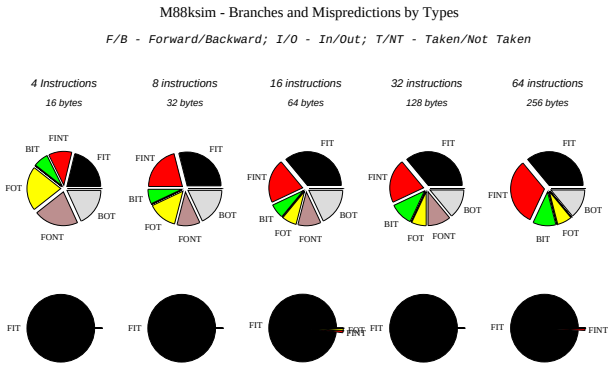

Figure 9.6: Short branches and Mispredictions - M88ksim

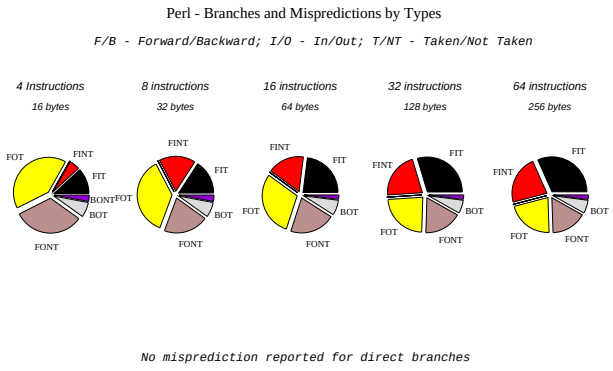

Figure 9.7: Short branches and Mispredictions - Perl

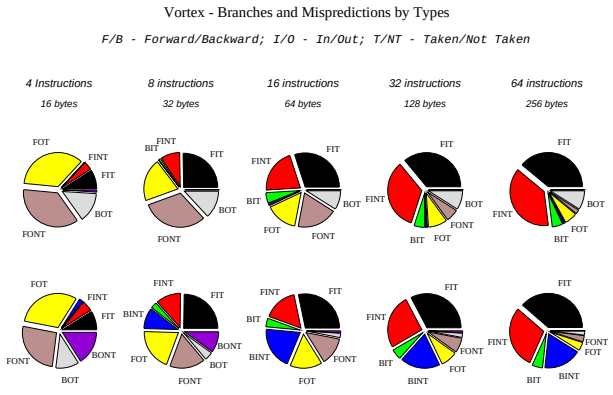

Figure 9.8: Short branches and Mispredictions - Vortex

References

[AGA00] AGARWAL, H. S.; MURUKKATHAMPOONDI, S. W.; BURGER, D. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[AHU98] AHUJA, Pritpal; SKADRON, Kevin; MARTONOSI, Margaret; CLARK, Douglas. Multipath Execution: Opportunities and Limits. In *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998.

[BUR96] BURGER, Doug; AUSTIN, Todd; BENNETT, Steve. Evaluating future microprocessors: The Simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin, CS Department, July 1996.

[CAL95] CALDER, Brad; GRUNWALD, Dirk. Next Cache Line and Set Prediction. In *Proceedings of 22nd Annual International Symposium on Computer Architecture.* June 1995.

[CHA95] CHANG, Po-Yung; HAO, Eric; PATT, Yale; Alternative Implementations of Hybrid Branch Predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture.* December, 1995.

[CHA99] CHAVES Filho, Eliseu; SANTOS, Francisco; SANTOS, Anna Dolejsi; NAVAUX, Philippe O. A.; SANTOS, Rafael R. MULFLUX: A Microarchitecture with Multiple Flows of Control. In: Protem-CC - Phase III Projects - International Evaluation, 2., 1999. *Pr oceedings...* Brasilia: CNPq, 1999. p. 146-176.

[CHE96] CHEN, I.-C.K.; COFFEY, J.T.; MUDGE, T. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[CHO99] CHOU, Yuan; FUNG, Jason; SHEN, John P. Reducing Branch Misprediction Penalties Via Dynamic Control Independence Detection. In *Proceedings of the 13th ACM International Conference on Supercomputing*, 1999.

[CON95] CONTE, Tom. et al. Optimization of Instruction Fetch Mechanism for High Issue Rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333-344, June 1995.

[EVE96] EVERS, Marius et al. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *Proceedings of the 23rd Annual International Symposium in Computer Architecture.* pp. 3-11, May 1996.

[FRI97] FRIENDLY, Daniel Holmes; PATEL, Sanjay; PATT, Yale N. Alternative Fetch And Issue Policies for the Trace Cache Fetch Mechanism. In *Proceedings of the 30th Annual International Symposium on Microarchitecture.* pp. 24-33, Research Triangle Pk, December 1997.

[FRI98] FRIENDLY, Daniel Holmes; PATEL, Sanjay; PATT, Yale N. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In *Proceedings of the 31th Annual International Symposium on Microarchitecture.* pp. 173-181, Dallas, December 1998.

[GWE96] GWENNAP, L. Digital 21264 Sets New Standard. *Microprocessor Report*, October 1996.

[GRU98] GRUNWALD, D; KLAUSER, A.; MANNE, S; PLESZKUN, A. Confidence Estimation for Speculative Control. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998. **Proceedings...** New York: ACM, 1998. p. 122-131.

[HEI96] HEIL, Timothy H.; SMITH, Jim *Selective Dual Path Execution.* Technical Report, University of Wisconsin, CS Department, 1996.

[HIL96] HILY, Sebastien; SEZNEC, A.; *Branch Prediction and Simultaneous Multithreading.* Technical Report No 997, IRISA, March 1996.

[JAC96] JACOBSEN, E.; ROTENBERG, E.; SMITH, J.E. Assigning confidence to conditional branch predictors. In *Proceedings of the 29th International Symposium on Microarchitectures.* 1996.

[JAC00] JACOBSON, Q.; SMITH, J.E. Trace Preconstruction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture.* pp 37-46, Vancouver, 2000.

[JOH91] JOHNSON, Mike. *Superscalar Microprocessor Design.* Englewood Cliffs: Prentince Hall, 1991.

[KES99] KESSLER, Richard E. The Alpha 21264 Microprocessor. **IEEE Micro**, New York, vol. 19, p 24-36, March/April 1999.

[KLA98a] KLAUSER, Artur; PAITHANKAR, Abhijit; GRUNWALD, Dirk. Selective Eager Execution on the Polypath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture.* pp. 250-259. June, 1998.

[KLA98b] KLAUSER, Artur; AUSTIN, Todd; GRUNWALD, Dirk; CALDER, Brad. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* pp. 278-285, Paris, October 1998.

[KLA99] KLAUSER, Artur; GRUNWALD, Dirk. Instruction Fetch Mechanisms for Multipath Execution Processors. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture.* Haifa, 1999.

[LAM92] LAM, Monica; WILSON, Robert. Limits of Control Flow on Parallelism. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46-57, 1992.

[LEE84] LEE, J; SMITH, A. J. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, pp. 6-22, 1984.

[MAH94] MAHLKE, Scott; et al. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture.* pp. 217-227, San Jose, California, November, 1994.

[MCF93] McFARLING, S. Combining Branch Predictors. WRL TN-36, Digital Western Research Lab. June, 1993.

[MIC98] MICHAUD, P.; SEZNEC, A.; JOURDAN, S.; SAINRAT, P. *Alternative Schemes for High-Bandwidth Instruction Fetch.* Technical Report No 1180, IRISA, March 1998.

[MIC99] MICHAUD, P.; SEZNEC, A.; JOURDAN, S.; SAINRAT, P. *Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors.* Technical Report No 1227, IRISA, January 1999.

[MIP95] MIPS Technologies Inc., MIPS R10000 Microprocessor User's Manual, Mountain View, CA, 1995.

[NAT97] *The National Technology Roadmap for Semiconductors.* SEMATECH, Austin, TX. 1997.

[PAN92] PAN, Shien-Tai et al. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. **ACM SIGPLAN Notices**, New York, v.27, n.9, p. 76-84, September 1992.

[PAT98] PATEL, Sanjay; EVERS, Marius; PATT, Yale. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture.* PP. 262-271, Barcelona, 1998.

[PEL94] PELEG, A.; WEISER, U. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line.* U.S. Patent Number 5,381,533, 1994.

[RAM00] RAMIREZ, Alex; LARRIBA-PEY, Josep; VALERO, Mateo. Trace Cache Redundancy: Red & Blue Traces. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[RAK00] RAKVIC, Ryan; BLACK, Bryan; SHEN, John P. Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance. In *Proceedings of the 27th Annual International Symposium on Computer Architecture.* pp. 47-58, Vancouver, 2000.

[ROT96] ROTENBERG, Eric; BENNET, Steve; SMITH, Jim. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. Madison: University of Wisconsin-Madison/Computer Science Departament, 1996. Technical Report.

[ROT99a] ROTENBERG, Eric; BENNET, Steve; SMITH, Jim. A Trace Cache Microarchitecture and Evaluation. IEEE Trans. on Computers, 48(2):111-120, Feb 1999.

[ROT99b] ROTENBERG, Eric.; JACOBSON, Quinn; SMITH, Jim. A Study of Control Independence in Superscalar Processors. In *Proceedings of 5th International Conference on High Performance Computer Architectures.* January, 1999.

[SAN98] SANTOS, Rafael R.; NAVAUX, Philippe O. A. Analysing a Multistreamed Superscalar Speculative Instruction Fetch Mechanism. In: EUROPAR. Southampton, October 1998. *Proceedings....* Lecture Notes in Computer Science, 1998.

[SAN99] SANTOS, Francisco; SANTOS, Rafael; SANTOS, Anna D. CHAVES Filho, Eliseu, NAVAUX, Philippe O. A. Performance Evaluation of a Microarchitecture With Multiple Flows of Control. In: Symposium on Computer Architecture and High Performance Computing, 11., 1999. *Proceedings...* Porto Alegre: SBC, 1999. p. 43-50.

[SAN01] SANTOS, Rafael; NAVAUX, Philippe; NEMIROVKSY. Mario. DCE: The Dynamic Conditional Execution Approach. In: Work in Progress Session of 7th IEEE International Symposium on High Performance Computer Architecture. Monterrey, Mexico, 2001.

[SEZ97] SEZNEC, A.; JOURDAN, S.; SAINRAT, P; MICHAUD, P. Multiple-block Ahead Branch Predictor. In *Proceedings of the 7th International Conference on Architectureal Suppport for Programming Languages and Operating Systems*, pp. 1997.

[SEZ99] SEZNEC, A.; MICHAUD, P. *De-Aliased Hybrid Branch Predictors.* Technical Report No 1229, IRISA, February 1999.

[SKA99] SKADRON, Kevin. *Characterizing and Removing Branch Mispredictions.* Princeton University, 1999. Ph.D. Thesis.

[SMI81] SMITH, James E. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture.* pp. 35-48, May 1981.

[SMI97] SMITH, James E.; VAJAPEYAM, Sriram. Trace Processors: Moving to Fourth-Generation. **Computer**, Los Alamitos, v.30, n.9, p.68-74, Sep. 1997.

[TAL95] TALCOTT, A.; NEMIROVSKY, M.; WOOD, R. The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques.* pp. 89-96, June 1995.

[TYS97] TYSON, G; LICK, K.; FARRENS, M. **Limited Dual Path Execution.** Technical Report CSE-TR 346-97, University of Michigan.

[UHT95] UHT, A.; SINDAGI, V. Disjoint Eager Execution: An Optimal Form for Speculative Execution. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 313-325, 1995.

[UHT97] UHT, A.; SINDAGI, V.; SOMANATHAN, S. *Branch Effect Reduction Techniques.* IEEE Computer. May 1997.

[WAL93] WALL, David. W. Limits of Instruction-Level Parallelism. WRL Technical Note TN-6, November 1993.

[WAL98] WALLACE, Steve; CALDER, Brad; TULLSEN, Dean. Threaded Multiple Path Execution. In *Proceedings of the 25th International Symposium on Computer Architecture.* June, 1998.

[WAL99] WALLACE, Steve; CALDER, Brad; TULLSEN, Dean. Instruction recycling on a Multiple-Path Processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture.* January, 1999.

[YEH91] YEH, Tse-Yu; PATT, Yale N. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51-61, 1991.

[YEH92] YEH, Tse-Yu; PATT, Yale N. Alternative Implementations of two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124-134, 1992.

[YEH93a] YEH, Tse-Yu; PATT, Yale N. 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, 1993.

[YEH93b] YEH, Tse-Yu; Marr, D.; PATT, Yale N. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proceedings of the 7th ACM International Conference on Supercomputing*, 1993.