UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL RAMOS DOS SANTOS

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

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Philippe O. A. Navaux
Advisor

Prof. Dr. Mario Nemirovisky
Coadvisor

Porto Alegre, August 2003

CIP – CATALOGING-IN-PUBLICATION

Santos, Rafael Ramos dos

DCE: The Dynamic Conditional Execution in a Multipath Control Independent Architecture / Rafael Ramos dos Santos. – Porto Alegre: PPGC da UFRGS, 2003.

128 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2003. Advisor: Philippe O. A. Navaux; Coadvisor: Mario Nemirovisky.

1. Branch Prediction. 2. Multipath Execution. 3. Dynamic Conditional Execution. 4. Dynamic Predication. I. Navaux, Philippe O. A.. II. Nemirovisky, Mario. III. Título.

“To my family and friends.”

ACKNOWLEDGMENTS

First, I would like to thank my wife, Tatiana. Her support was immeasurable. I cannot fully express how important she is. Throughout the past five years she has helped me in all ways and has given to me every bit of attention and comfort possible. Without her, I doubt I would have achieved this.

My parents, who have provided the incentive and financial support that get me started. I have so much to thank them. And my family and in laws, to all special moments and because they are all I most care in life.

I would like to thank Professor Philippe Navaux for providing all the opportunities a student need. I have been working with him for 10 long years now and his support was certainly essential to take me to this position.

In the recent years I had the opportunity to share thoughts with Mario Nemirovsky who has dedicated an enormous amount of time helping me and giving me lessons in computer architecture while managing three startup companies. I have most respect for him.

I have also to thank Adam Talcott and Wayne Yamamoto for introducing me to Mario Nemirvosky and for inspiring my initial studies in this topic. I enjoy working closely with them nowadays.

My colleagues and friends from the APSE/GPPD, Tatiana Santos, Mauricio Pilla, Ronaldo Goncalves, Guilherme Pizzol, Rafael Sagula and many others for the constant discussions, insights and feedbacks in my work.

I want to thank Professor Eliseu Chaves Filho for inviting me to be part of the MULFLUX project back in 1997 which ultimately inspired this work and for the many discussions over this topic.

I want to thank Professor Joel Ferguson from UCSC Jack Baskin School of Engineering who supported me for two years while I was a researcher at his department.

Many thanks to Carolee Thompson (my manager), Craig Johnson and Babu Turumella from Sun Microsystems for their understanding and support.

Thanks to Artur Klauser for providing his simulator which leveraged most of the development in the DCE parser. Eric Rotemberg and his students for sharing insights and preliminary results in CIDI.

Thanks also to the Labtec/Dell cluster administration from UFRGS for providing the computing resources to run my simulations and for CNPQ and UNISC for partially sponsoring the development of this work.

I cannot forget my friends Mumu, Daniela, Marcela, Andreas, Nedja, Ian, Lara, Enric, Eby, Sergio, Koichi, Ricardo, Pedone, Pascale, Joni, Andrea, Harold, Venu and Shantanu for their friendship and companionship. I really appreciated the chance to be with them and of course I enjoyed the parties, barbecues, guitar sessions, karaoke and beer drinking.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ILP	Instruction Level Parallelism
CI	Control Independent
DI	Data Independent
CIDI	Control Independent and Data Independent
BTB	Branch Target Buffer
DCE	Dynamic Conditional Execution
TTB	Target Tagid Buffer

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This thesis presents DCE, or Dynamic Conditional Execution, as an alternative to reduce the cost of mispredicted branches. The basic idea is to fetch all paths produced by a branch that obey certain restrictions regarding complexity and size. As a result, a smaller number of predictions is performed, and therefore, a lesser number of branches are mispredicted.

DCE fetches through selected branches avoiding disruptions in the fetch flow when these branches are fetched. Both paths of selected branches are executed but only the correct path commits.

In this thesis we propose an architecture to execute multiple paths of selected branches. Branches are selected based on the size and other conditions. Simple and complex branches can be dynamically predicated without requiring a special instruction set nor special compiler optimizations.

Furthermore, a technique to reduce part of the overhead generated by the execution of multiple paths is proposed. The performance achieved reaches levels of up to 12% when comparing a Local predictor used in DCE against a Global predictor used in the reference machine. When both machines use a Local predictor, the speedup is increased by an average of 3-3.5%.

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

DCE: Execução Dinâmica Condicional em uma Arquitetura de Múltiplos Fluxos com Independência de Controle

RESUMO

Esta tese apresenta DCE, ou Execução Dinâmica Condicional, como uma alternativa para reduzir o custo da previsão incorreta de desvios. A idéia básica do modelo apresentado é buscar e executar todos os caminhos de desvios que obedecem à certas restrições no que diz respeito a complexidade e tamanho. Como resultado, tem-se um número menor de desvios sendo previstos e consequentemente um número menor de desvios previstos incorretamente.

DCE busca todos os caminhos dos desvios selecionados evitando quebras no fluxo de busca quando estes desvios são buscados. Os caminhos buscados dos desvios selecionados são então executados mas somente o caminho correto é completado.

Nesta tese nós propomos uma arquitetura para executar múltiplos caminhos dos desvios selecionados. A seleção dos desvios ocorre baseada no tamanho do desvio e em outras condições. A seleção de desvios simples e complexos permite a predicação dinâmica destes desvios sem a necessidade da existência de um conjunto específico de instruções nem otimizações especiais por parte do compilador.

Além disso, é proposta também uma técnica para reduzir a sobrecarga gerada pela execução dos múltiplos caminhos dos desvios selecionados. O desempenho alcançado atinge níveis de até 12% quando um previsor de desvios Local 'e usado no DCE e um previsor Global é usado na máquina de referência. Quando ambas as máquinas empregam previsão Local, há um aumento de desempenho da ordem de 3-3.5%.

Palavras-chave: Previsão de Desvios, Execução Multipath, Execução Dinâmica Condicional, Predicação Dinâmica.

1 INTRODUCTION

Instruction supply for high performance superscalar microprocessors is a demanding task which relies essentially on the ability of the processor to predict the outcome of conditional branches.

The branch prediction technique is widely used to predict the behavior of conditional branches in order to allow continuous fetching of instructions even when the correct flow of control is not known. However, branch misprediction and fetch disruption degrade significantly the performance of the processor as pipelines get deeper.

This work proposes a method for reducing penalties associated with branch mispredictions by using the available fetch bandwidth to supply instructions from both paths of selected branches.

The branches that benefit from this fetch technique, are executed eagerly thus not requiring to be predicted. By reducing the number of predicted branches it is possible to reduce the absolute number of mispredicted branches thus reducing also the overall misprediction penalty while not requiring additional fetch bandwidth. In fact, the predictor engine may be slightly simpler since less branches need to be predict.

1.1 Rationale

Future microprocessors will face more serious limitations concerning instruction supply as actual branch predictors are likely to have approached their best prediction rates.

The only apparent way to increase instruction supply is to feed the processor with multiple basic blocks per cycle, in single thread machines. Since the average basic block's size for common applications is in the range of 3-4 instructions, a processor which is targeted to execute for example 16 instructions per cycle would require an average of 4-5 basic blocks of useful instructions per cycle to accommodate this level of performance and parallelism. Perhaps more than 4 branches would have to predicted per cycle in order to achieve this goal. The complexity of a branch predictor of this caliber would certainly impose serious constraints from the implementation point of view.

An effective reduction of the number of predicted branches leads to a reduction of the overall misprediction penalties since less mispredictions would occur (SANTOS; NEMIROVSKY, 2001; SANTOS; NAVAUX; NEMIROVSKY, 2001). Another benefit would be to reduce the size of the tables necessary to predict these branches.

Some complex compiler techniques allow to increase the size of basic blocks by moving instructions around them. Such techniques are sometimes not efficient because they miss some dynamic information and therefore are not always capable of removing some branches.

An interesting approach is to execute both paths of a branch. Executing both paths eliminate the need to predict the behavior of the branch. The immediate result is less mispredictions and consequently less penalties added to the performance.

1.2 Problem Statement

The question is: how to reduce the number of predicted branches ? A branch is a two-way construct where there is a condition to be evaluate and a path to be followed according to the condition. Hence, a conditional branch has only two possible paths: taken or not taken.

When a branch is predicted, only one path is followed thus creating the chance to mispredict the branch and execute instructions down the wrong path. Recovering after a mispredicted branch is relatively simple but the penalty resides in the fact that several instructions may be accessed and executed while the result of the branch is computed and thus have to be squashed since the context generated is not valid.

When both paths of a branch are executed, it is not necessary to predict the branch itself. Although the problem of deciding what path has to be executed is eliminated, another problem arises when the branch converges into a common path.

A conditional branch always converge into a merge point except when the program exits through one of the paths. A merge point thus starts a piece of code that is common to both paths of a branch also known as Control Independent (CI) path.

The idea of executing branches in a eager manner is interesting under the assumption that the machine is able to handle all possible combinations of the branch and all paths spawn following the initial branch. For one branch there are only two combinations: the taken and the not taken. The number of combinations increases though when new branches are executed eagerly, while there are older outstanding branches in flight.

When the two paths of a branch merge, the CI path must be replicated for each of the combinations in order to guarantee correct data dependency. This introduces a major concern on implementing multiple path execution schemes because the replication suggests an exponential growth of the number of paths being executed as the number of branches increases.

Therefore, a kind of selection is necessary to avoid excessive growth. Several models were proposed basing the selection of branches for eager execution on confidence mechanisms. If the predicted branch has low confidence then it is executed eagerly. Otherwise the processing is based on conventional prediction.

In our study we propose to investigate a form of multipath which exploits the locality of some branches. By looking into the position of the taken target we decide whether a branch should be executed eagerly or not.

We believe that an optimal form of multipath could benefit from locality and:

1. improve fetch efficiency
2. simplify branch prediction mechanism
3. reduce necessary fetch throughput
4. reduce overall misprediction occurrence, and
5. increase performance

1.3 Purpose of Study

The present study is intended to investigate the performance and feasibility of a multipath mechanism with selection based on branch spatial locality. The investigation firstly addresses whether the mechanism can delivery increased performance and secondly, what features would have to be added to the reference architecture and how the architecture would perform.

Below are the objectives of this study:

1. quantify locality and patterns of branches under different circumstances (e.g. distance, cache position)
2. analyze the distribution of misprediction under the circumstances considered above
3. quantify the improvement possible to be reached under optimal assumptions (upper bound analysis)
4. determinate optimal distances according to the depth of the pipeline (this will impact the number of paths can/need-to be execute to guarantee certain performance indexes
5. determine the basic architecture capable of supporting the features stated, and
6. evaluate the architecture

1.4 Hypotheses

We believe that:

- there are a significant number of branches which have very near targets
- a significant portion of mispredictions fall into this class of branches
- short branches (either intra-block or not) and their both paths can be fetched easily by accessing the memory sequentially
- useful dispatch rate will increase since fetch disruption can be diminished
- fetch buffer collapsing is not necessary since both paths are executed
- less branches are predicted and thus branch prediction tables may be smaller

We expect to:

- reduce misprediction associated with short branches
- simplify the fetch by imposing a more natural sequential access
- reduce the size and complexity of branch prediction tables

We want to understand better:

- what is the impact of overloading the execution core (wrong path overhead)
- how many paths are necessary to guarantee certain performance
- how many paths and instructions will be fetched and executed for a given distance (replication overhead)

1.5 Outline

In this thesis our contribution is two-fold. First, we analyze and profile branch mispredictions and we propose a dynamic conditional execution processor that is capable of executing eagerly multiple instructions from disjoint points in the program. We show how to build a processor that can execute instructions from multiple paths of simple and complex branches.

Second, we evaluate the performance of this processor by executing multiple benchmarks over several different configurations analyzing the potentials and drawbacks of this approach. We also propose an optimization technique and provide insight for future research to reduce overhead generate to compensate the aggravation of data dependences introduced by the eager execution.

Related works and background are presented in Chapter 2 and in Chapter 3, we present an analysis of the impact of branch mispredictions on the machine and fetch efficiency in wide-issue superscalar machines.

Chapter 4 presents an analyzes based on branch profiling extracted from several different benchmarks. We provide distributions of branch mispredictions based on branch distances.

The DCE architecture is defined in Chapter 5. We describe how each stage of the pipeline is implemented and discuss the advantages of each approach adopted. The experimental environment and configurations simulated are detailed in Chapter 6 and Chapter 7 presents the performance evaluation of DCE.

At last, Chapter 8 presents the conclusion and a discussion of future works.

2 BACKGROUND

2.1 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 (LEE; SMITH, 1984; YEH; PATT, 1991; LAM; WILSON, 1992; WALL, 1993; CHAVES FILHO; FERNANDES, 1997) and can only delivery up to one basic block of useful instructions to the execution engine, if and when 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 2.1. Either reducing/eliminating taken branches or predicting multiples branches per cycle is crucial to increase fetch throughput.

Several schemes have been proposed to reduce the negative effect of taken branches (UHT; SINDAGI; SOMANATHAN, 1997). An effective technique to increase fetch bandwidth must: (i) predict branches successfully and (ii) fetch multiple basic blocks per cycle.

2.1.1 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 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 behavior of future branches.

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 (SKADRON, 1999).

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 (SKADRON, 1999) 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

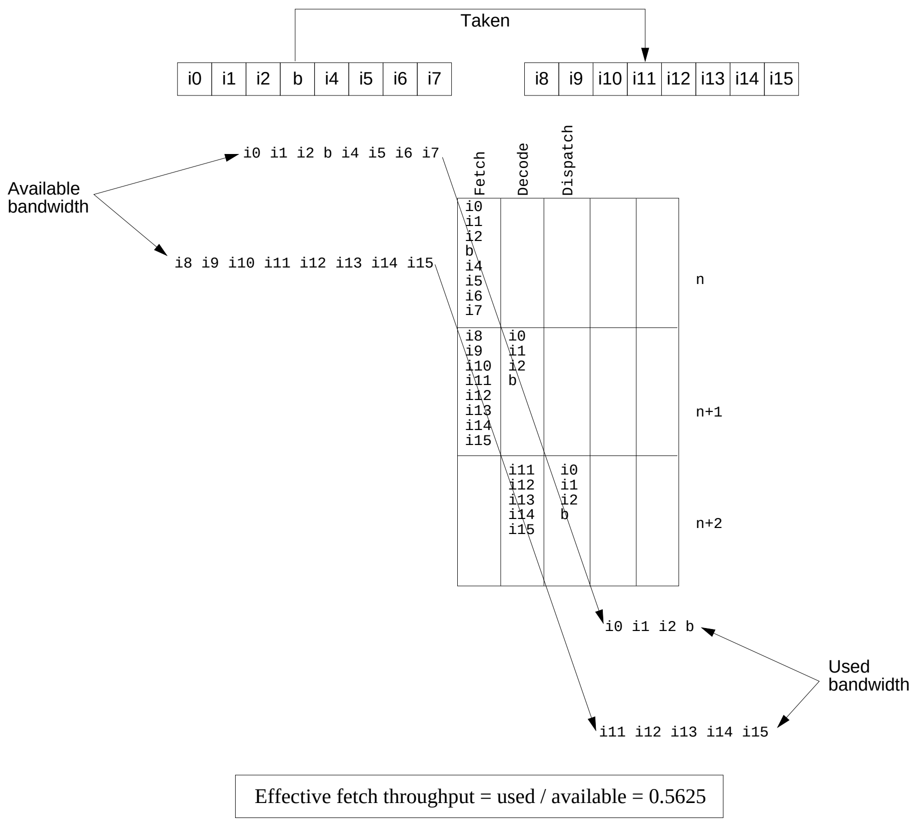


Figure 2.1: Instruction Flow Disruption

bimodal predictor was proposed by Smith (SMITH, 1981) 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 (YEH; PATT, 1991) 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 behavior 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 behavior of other recent branches. This is providential for predicting sequences of correlated branches (PAN; SO; RAHMEH, 1992; YEH; PATT, 1992, 1993).

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. (TALCOTT; NEMIROVSKY; WOOD, 1995).

The aliasing problem can be alleviated by combining the history bits with some bits from the branch's address. McFarling (MCFARLING, 1993) 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 behaviors, 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 (MCFARLING, 1993; CHANG; HAO; PATT, 1995; EVERS; AL., 1996).

An example was described by Kessler (KESSLER, 1999). He described the predictor used in the Alpha 21264 (GWENNAP, 1996). The processor employs a hybrid scheme which selects a prediction from a local and global predictors achieving from 90% to almost 100% accuracy in some applications. Seznec also proposed a ”De-aliased”global predictor which achieves the same prediction accuracy level of *gshare* or *gselect* (YEH; PATT, 1993) using less than half of the transistor budget (SEZNEC; MICHAUD, 1999).

2.1.2 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.

Yeh et al. (YEH; MARR; PATT, 1993) 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.

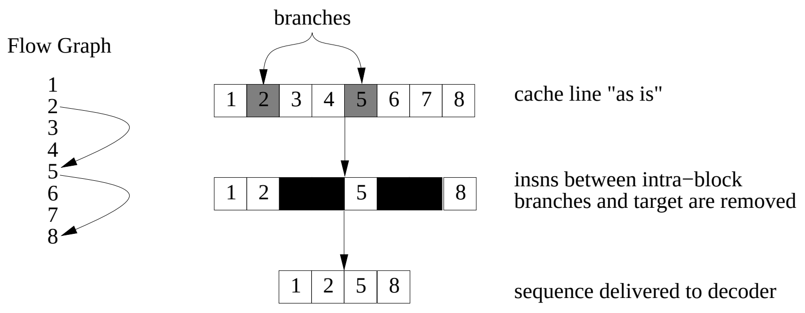

branches
Flow Graph
1
2
3
4
5
6
7
8
cache line "as is"
insns between intra−block branches and target are removed
sequence delivered to decoder

Figure 2.2: Collapsing Buffer Scheme

Conte et al. (CONTE et al., 1995) 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 2.2). 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.

Essentially, these two schemes (YEH; MARR; PATT, 1993; CONTE et al., 1995) 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. (ROTENBERG; BENNET; SMITH, 1996) 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.

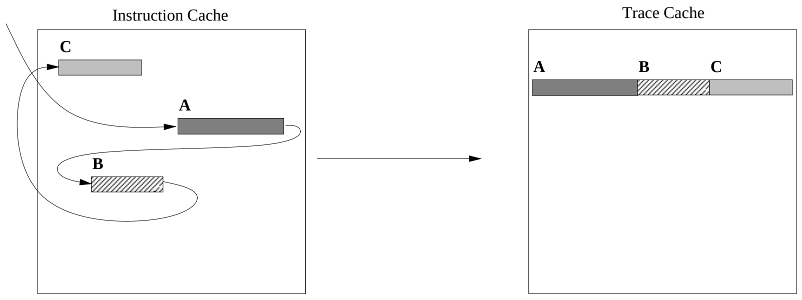


Figure 2.3: Dynamic Trace and Trace Cache

Figure 2.3 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. (FRIENDLY; PATEL; PATT, 1997) 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 (FRIENDLY; PATEL; PATT, 1997). 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. (PATEL; EVERS; PATT, 1998) 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 (PATEL; EVERS; PATT, 1998).

Friendly et al. (FRIENDLY; PATEL; PATT, 1998) 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 effecting 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-up 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 (ROTENBERG; BENNETT; SMITH, 1999). 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. (RAMIREZ; LARRIBA-PEY; VALERO, 2000) 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 (JACOBSON; SMITH, 2000) 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. (SEZNEC et al., 1997) 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. (MICHAUD et al., 1998) compared Pros and Cons of TBA (*Two-Block Ahead*) against the Trace Cache (ROTENBERG; BENNET; SMITH, 1996). 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. (MICHAUD et al., 1999) 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 (MICHAUD et al., 1999; CHANG; HAO; PATT, 1995).

A second alternative is to execute both paths of a conditional branch rather than predicted the most likely taken path. Some schemes spawn both paths of every conditional branch at the expense of a exponential overhead growth as more branches are very likely found in each new path created (UHT; SINDAGI, 1995). Other approaches spawn both paths of only certain branches (HEIL; SMITH, 1996; AHUJA et al., 1998; KLAUSER; PAITHANKAR; GRUNWALD, 1998) based on a confidence predictor (JACOBSEN; ROTENBERG; SMITH, 1999). 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.

2.2 Branch Misprediction and Multipath

As presented previously most of the current research to increase instruction supply rely on branch prediction. Their intention is basically allow more than one basic block to feed the decoder thus increasing instruction throughput. Obviously, performance depends on predictor's accuracy 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 93% or more. Despite that great accuracy, few mispredictions are enough to harm the performance, because the penalty is indeed 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 (LAM; WILSON, 1992; SANTOS, 1997; CHAVES FILHO et al., 1999; SANTOS et al., 1999).

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.

2.2.1 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. To reduce the negative branch effects and minimize the dependencies he proposed DEE - Disjoint Eager Execution (UHT; SINDAGI, 1995). DEE is based on the idea of executing multiple paths simultaneously in order to avoid mispredictions and misspeculations, completely or partially. 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 likely branch paths are provided with resources. DEE is based on the cumulative probability of a branch path is executed. Instead of fetching and executing 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 the eager execution's high cost (UHT; SINDAGI, 1995).

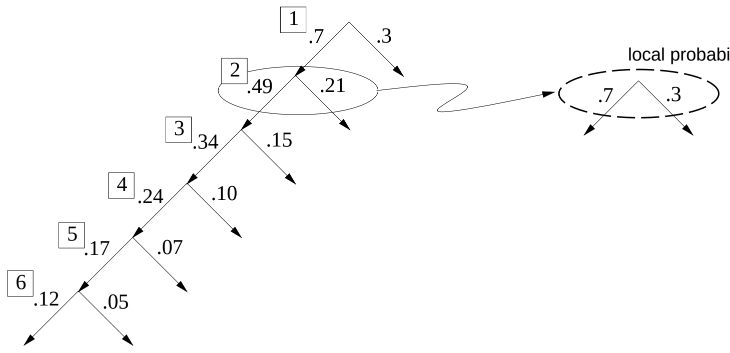


Figure 2.4: Single Path Architecture with Cumulative Branch Probabilities

Figure 2.4 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 2.4.

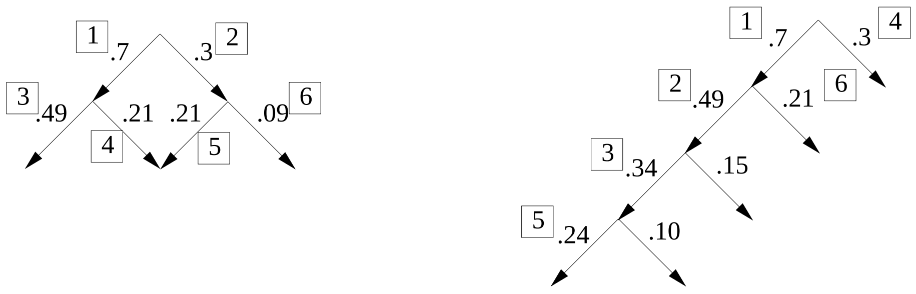


Figure 2.5: Eager and DEE Multiple Path Architectures

Figure 2.5 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 2.4 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 2.5 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 dynamic nature of the conceptual model.

2.2.2 Selective Dual Path Execution

Another interesting work, presented by Heil and Smith (HEIL; SMITH, 1996), 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. In other words, the most likely path receives 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 (HEIL; SMITH, 1996). 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.

2.2.3 Dynamic Predication

There is few work done on dynamic predication. The most known was done by Klauser (KLAUSER et al., 1998) whom proposed a dynamic predication mechanism for non-predicated instruction set architectures.

Their mechanism allows to execute both paths of simple hammocks while not requiring special instruction set nor compiler optimizations. The mechanism dynamically predicates instructions sequences derived from simple hammocks. A simple hammock is a conditional forward branch that has not nested branches in both the taken and the not taken paths.

The authors classify branches into four distinct groups but select only simple hammocks for predication. Simple hammocks are branches that have no nested branches and contain a single basic block in each of the paths. They show that about 11% of the mispredictions fall into this class of branches.

Their machine applies tags to predicated paths and renames instructions accordingly. At the join point, to resolve inter-path false dependencies, the machine injects conditional moves.

Conditional moves are injected dynamically at the join point to block the issue of instructions from the same data chain of the predicated paths. When the branch resolves, the conditional moves can be issued to copy the data from the correct physical register to the correct source register. Thus, the original instruction that uses the respective register becomes ready for issue only after the conditional move instruction executes.

These injected instructions will block the issue of instructions at the join point only if they depend on data produced in one of the predicated paths. By doing this, they guarantee that a dependent instruction is only issued when the correct data is available.

In their dynamic predication architecture they observed an average of approximately 5% improvement over a conventional machine by predicating simple hammocks only and using a size-based static selection mechanism.

2.3 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.

Though 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.

2.3.1 An Introductory Example

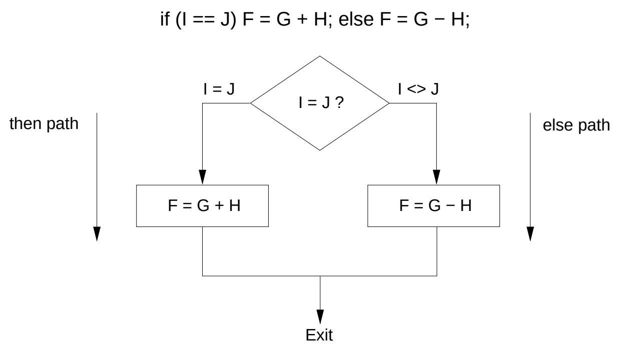


Figure 2.6: A “C” *if* Statement

Figure 2.6 is a simple piece of code written in C language. 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 2.7 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 2.8 shows how the code looks like and the position of the instructions inside an 8 instructions memory cache line.

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.

```
       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 2.7: The Compiled MIPS Assembly

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.

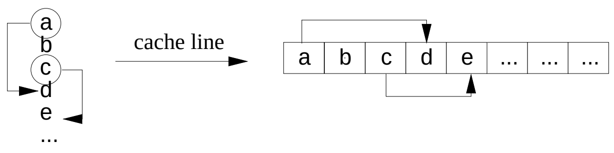
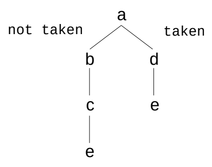


Figure 2.8: The Symbolic Example

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 2.9 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*.



	first access	second access
not taken:	a b c	e
taken:	a	d e

Figure 2.9: 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 2.8. 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 2.10, three accesses to the same cache line are necessary to bring instructions for both taken and not taken paths of branch *a*.

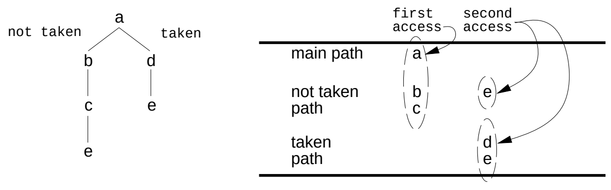


Figure 2.10: The Eager Multiple Paths Version

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 2.8, 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.

2.3.2 An Extended Example

The examples showed previously intended to maximize the understanding of multi-path 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 2.11 extends the previous example introducing another conditional branch to the case.

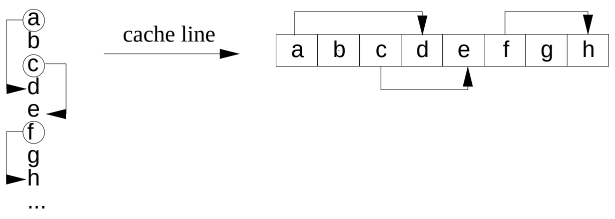


Figure 2.11: An Extended Case

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 2.12 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 repli-

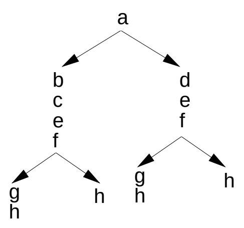


Figure 2.12: The Result: an exponential growth

cated again, as occurred in the example presented in Figure 2.10. 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*.

2.3.3 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 2.10. 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.

2.3.4 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 2.13 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

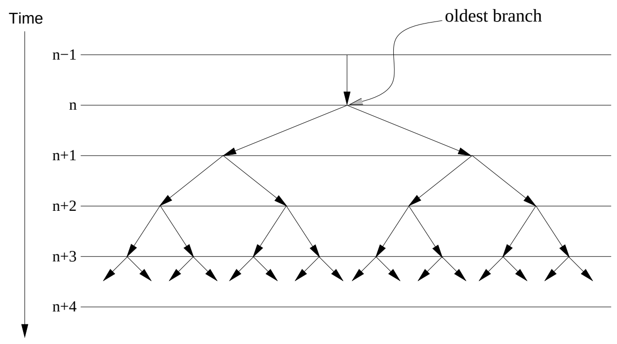


Figure 2.13: The Dynamic Tree After the Oldest Branch is Fetched

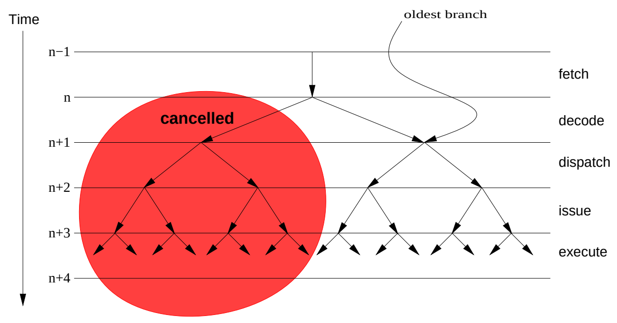


Figure 2.14: The Dynamic Tree After the Oldest Branch is Resolved

oldest branch takes more than four cycles to resolve, the number of active paths may increase.

Figure 2.14 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.

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).

3 THE ROLE OF BRANCHES AND BRANCH PREDICTION

In this chapter we discuss the role of branch prediction in current and future microprocessors. We present the benchmarks characterization and a reference architecture where we simulate several experiments. We address the misprediction effects in the efficiency of the fetch engine and its impact in the performance.

3.1 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.

Table 3.1 shows each benchmark and the percentage 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.

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.

Figure 3.1 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.

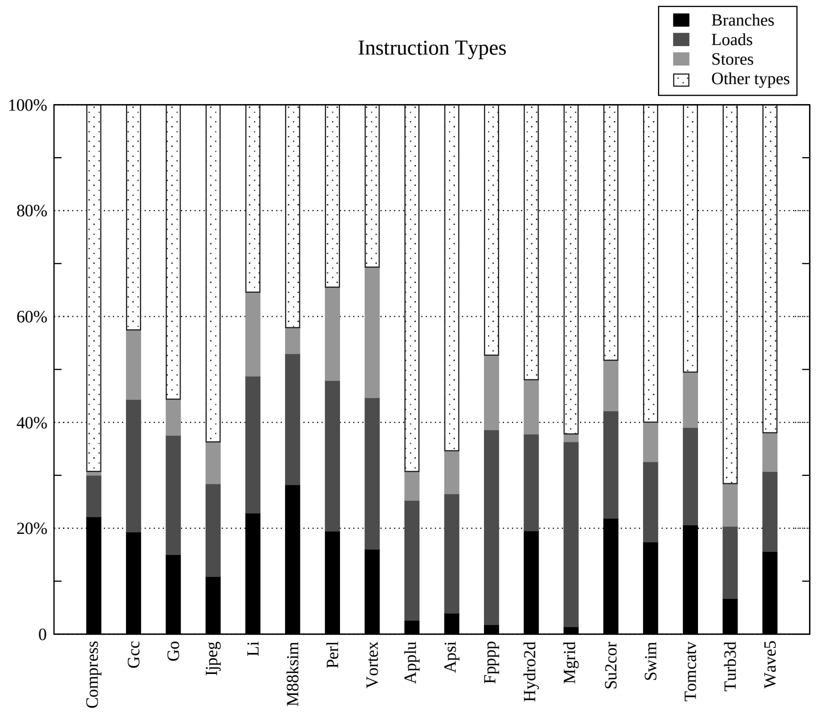


Figure 3.1: Classes of Instructions

Table 3.1: 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

3.2 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 3.2). 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;

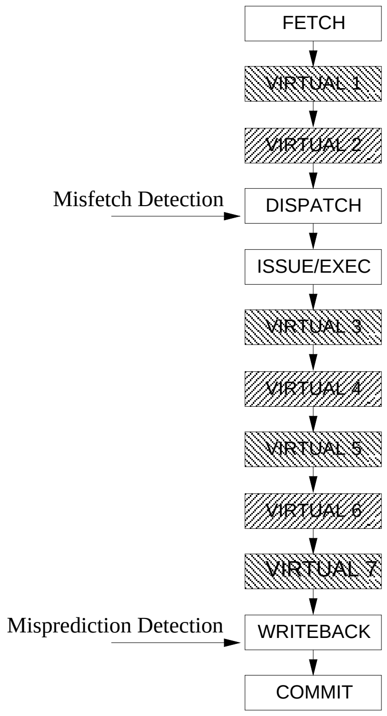


Figure 3.2: Pipeline Structure

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 (BURGER; AUSTIN; BENNET, 1996).

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.3 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.

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 3.3 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

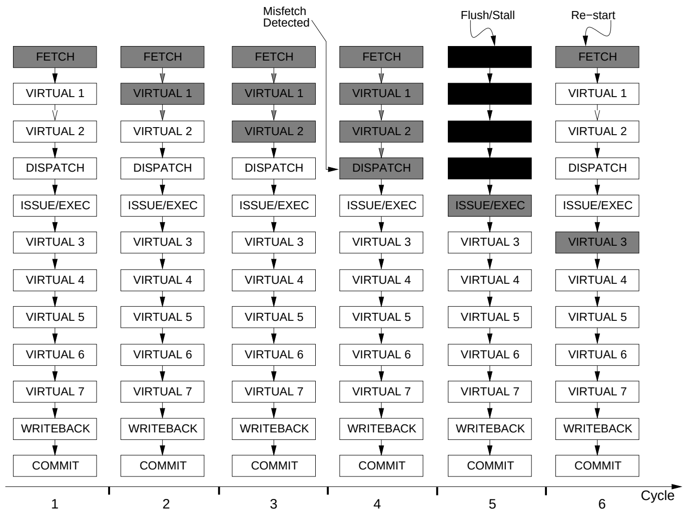


Figure 3.3: Misfetch Penalty

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.

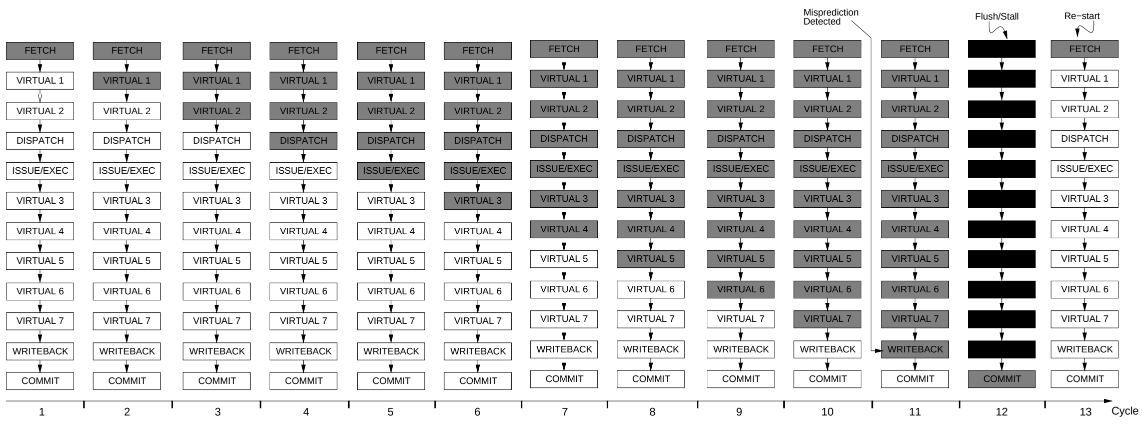


Figure 3.4: Misprediction Penalty

Figure 3.4 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 3.3 and 3.4 showed the penalties associated with misfetches and mispredictions as defined in the reference architecture used hereafter.

3.4 IPC - Instructions per Cycle

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

$$IPC = \frac{number\ of\ instructions\ committed}{cycles} \tag{3.1}$$

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.

The graph on Figure 3.5 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.

3.5 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.

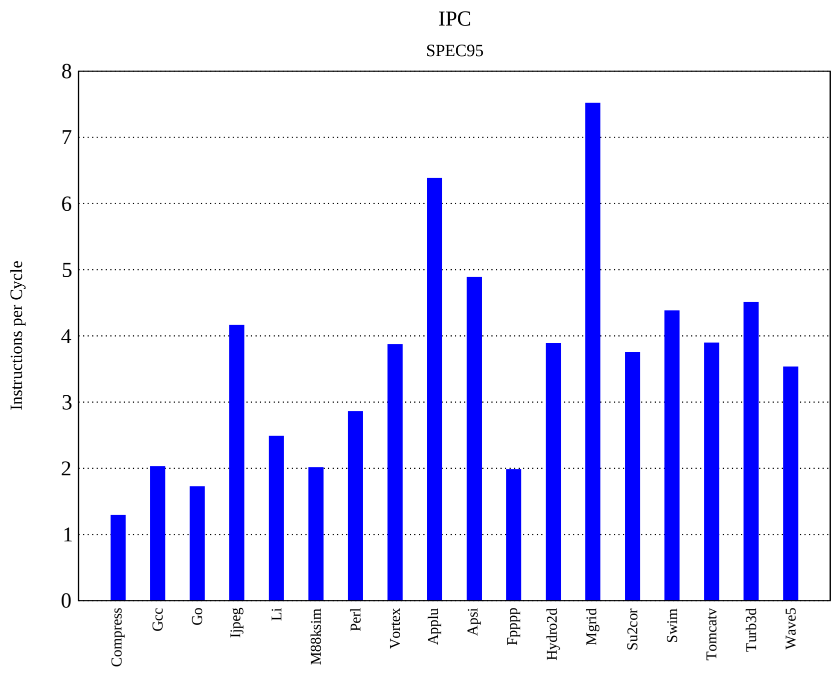


Figure 3.5: IPC of SPEC95

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 effect 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.

3.6 Fetch Performance

3.6.1 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 stall latency will depend on the subsequent stages of the pipeline because they need to consume the instructions that are obstructing the fetch.

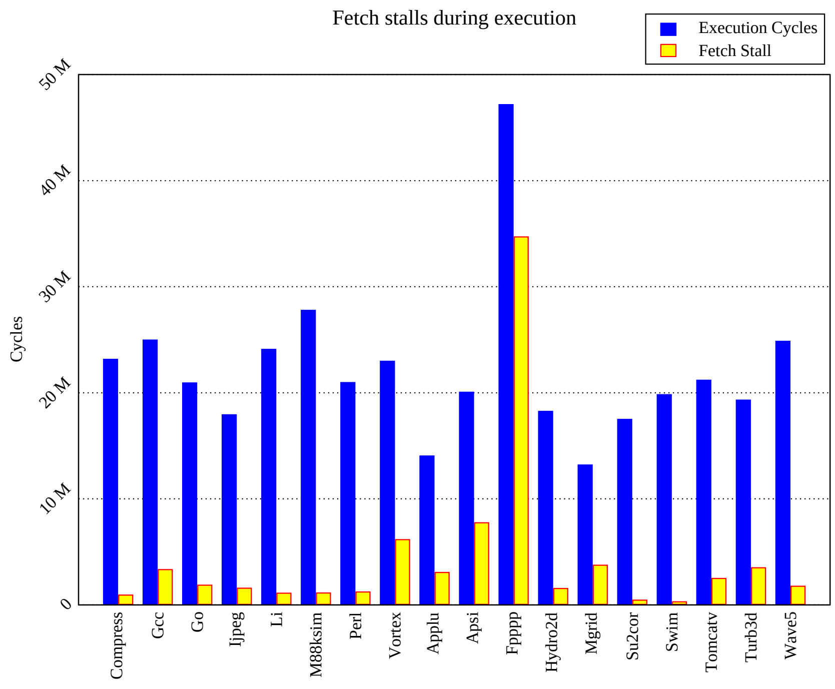


Figure 3.6: Fetch Stalls

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 3.6 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 3.7 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.

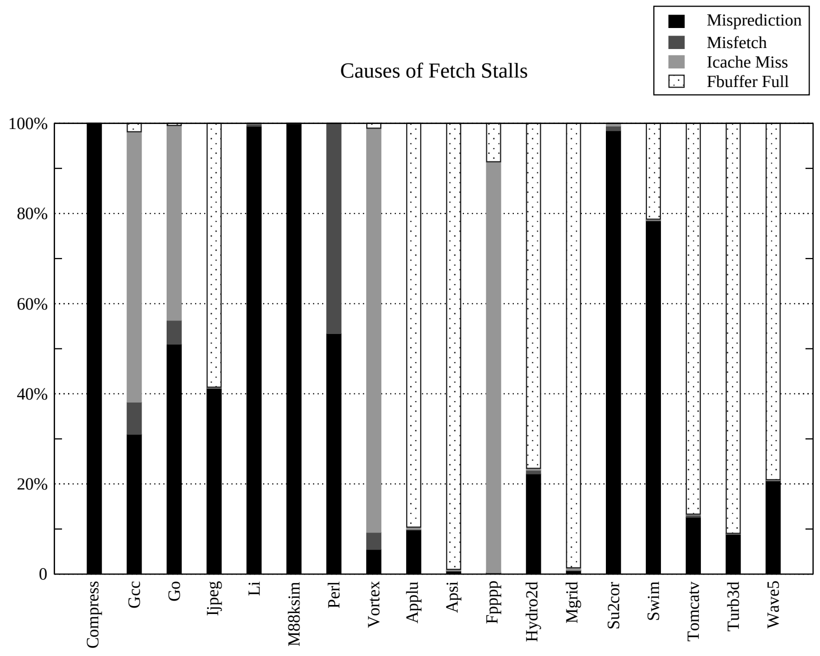


Figure 3.7: Causes of Stalls

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.

Fetch stalls caused by fetch buffer full despite causing a stall they actually show that the fetch engine is capable of processing faster than the execution core and one could in fact point that as a positive aspect of the fetch engine. A fetch buffer full occurs when there is no space to store fetched instructions so the fetch has to stop. It is has an effect in the fetch engine but it is actually sourced in the back end engine.

3.6.2 Fetch Loss

Fetch loss, is defined as the number of cycles where the fetch unit was stalled or it performed work that was later squashed. In other words, fetch loss is the number of useless cycles from the fetch point of view.

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 the overall performance of the fetch engine is mainly affected by mispredictions and that even a few number of mispredictions can badly harm the performance of the fetch.

In the previous figures we showed the cycles were the fetch was inactive. In Figure 3.8 we are showing all cycles lost due to the four events named: 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 afterward.

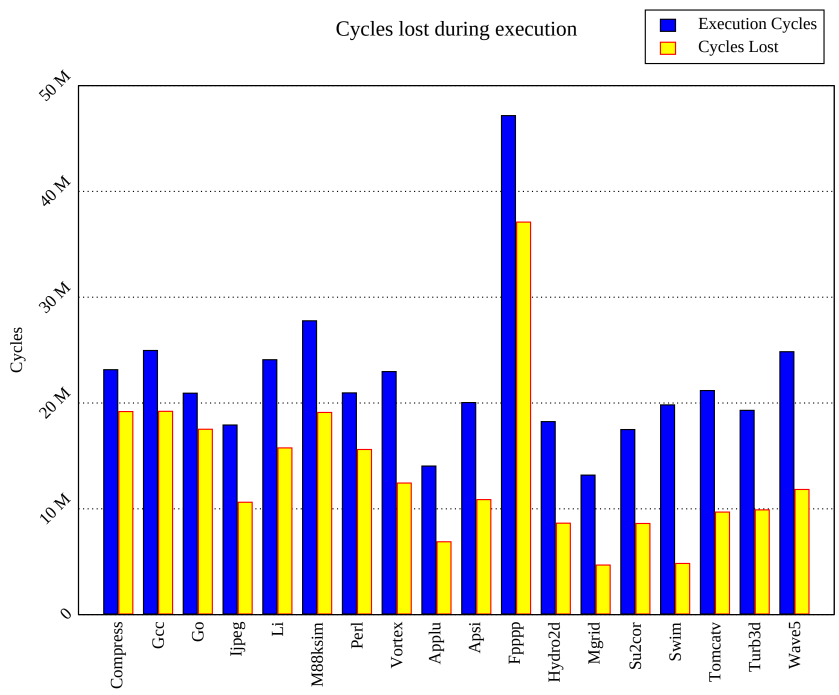


Figure 3.8: Cycles Lost

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 *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 3.2 summarizes the prediction accuracy and misprediction rate for each benchmark.

Table 3.2: 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

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 3.2 we can see that prediction accuracy varies from 84.11% through 99.36% which is in average a good accuracy for the prediction.

Figure 3.9 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

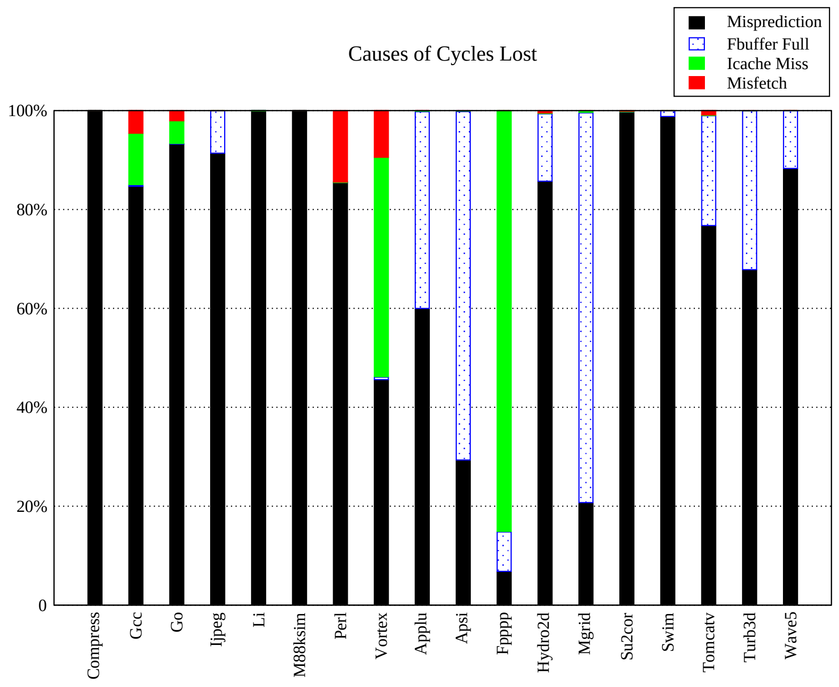


Figure 3.9: Causes of Loss

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.

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 effect 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.

3.6.3 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 3.10 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.

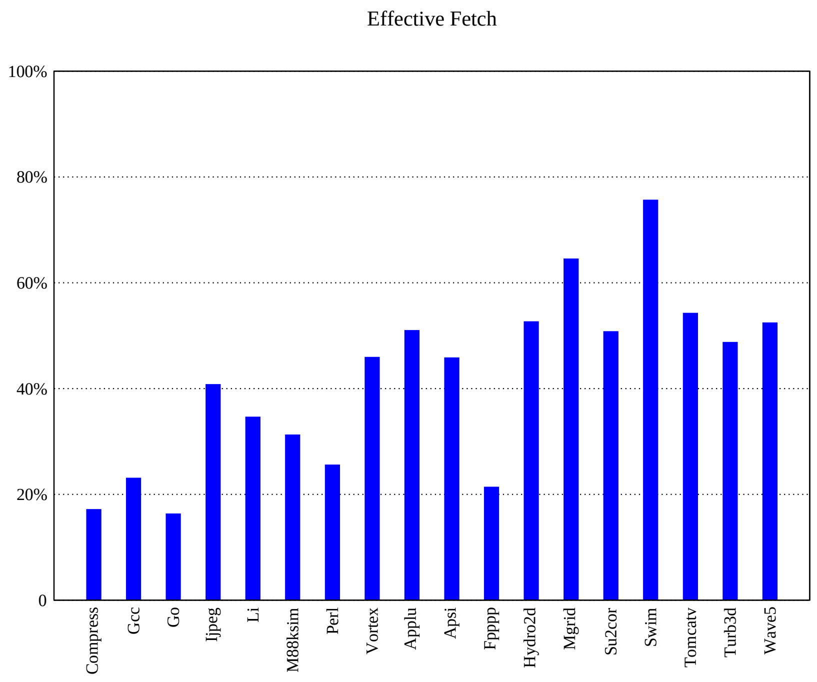


Figure 3.10: Effective Fetch

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 3.10 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.

3.7 Impact on Effi ciency

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 problems 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.

3.7.1 Overall Statistics

Tables 3.3 and 3.4 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 3.3: 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 3.3 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 3.4 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

Table 3.4: 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

4.47 instructions per cycle where integer benchmarks achieved only 2.55.

Another interesting point is that for integer benchmarks, the accuracy 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 difference, the performance of the float point benchmarks is limited by mispredictions and Icache misses too. We sorted the benchmarks accordingly to their the misprediction penalty, Table 3.4. 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. This benchmark reported the highest fetch stall rate among all benchmarks. The rest of the float point benchmarks did not present significant Icache miss rates.

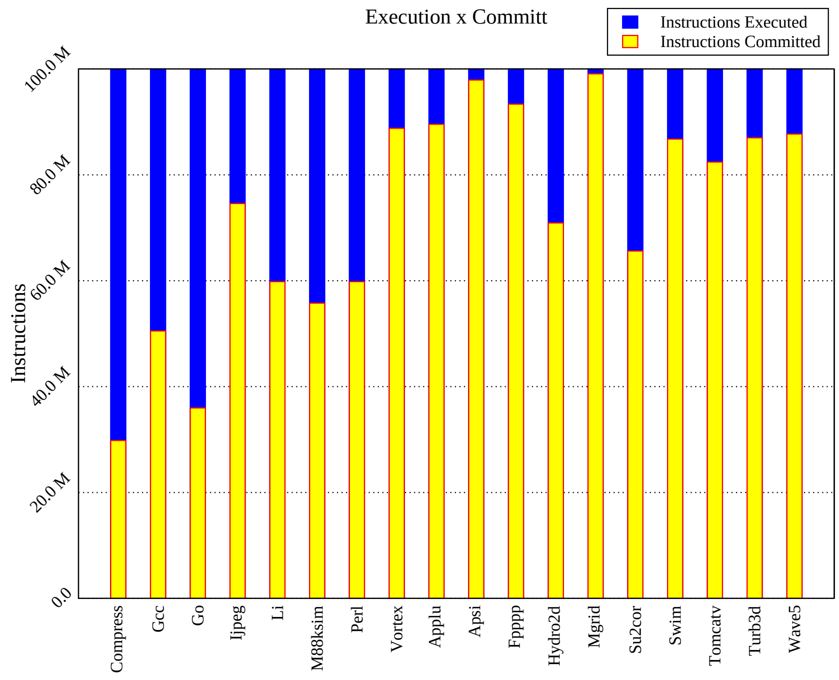


Figure 3.11: Machine Efficiency

Figure 3.11 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 the case of *Compress*, only 29.82% of instructions executed are finally committed. This represents a major penalty not only by squashing useless instructions, but by using resources to execute them as well. In the best case, on the other hand, we see 99.09% of executed instructions being 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 as to why the average IPC is as twice as high as the one shown by integer benchmarks.

3.8 Perfect Branch Prediction and its Impact on Performance

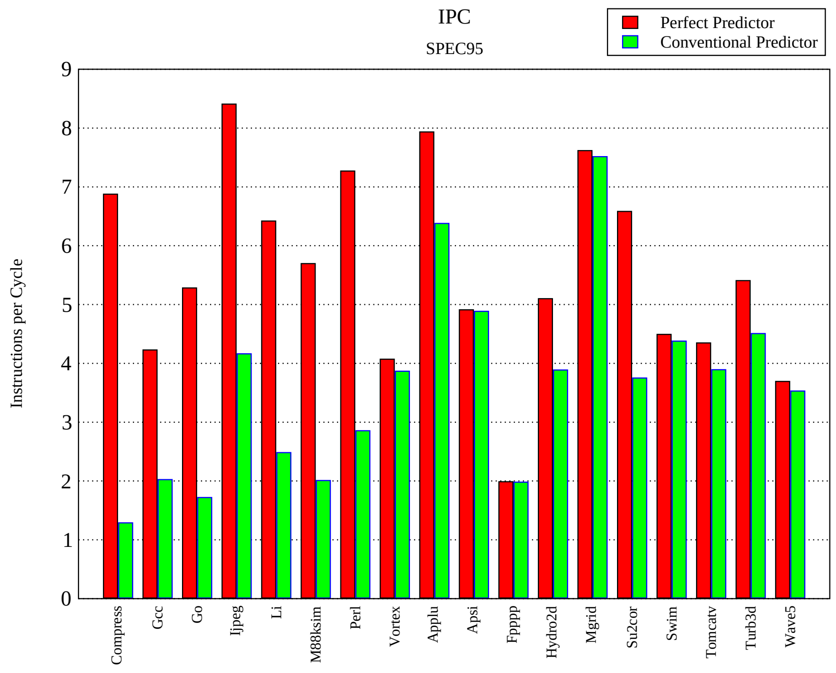


Figure 3.12: Machine Perfomances with Perfect Prediction and Conventional Prediction

Figure 3.12 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.

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.

4 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.

4.1 Branch Direction and Outcome

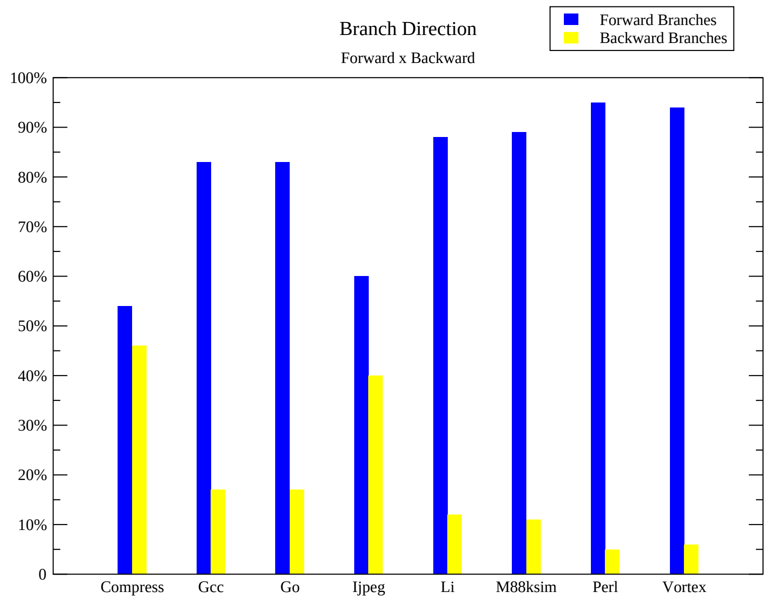


Figure 4.1: Branch Direction

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 4.1 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.

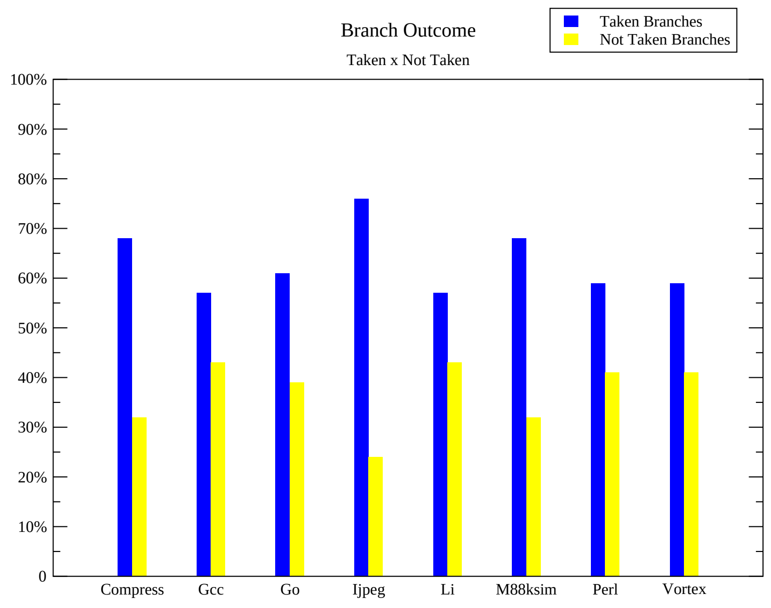


Figure 4.2: Branch Outcome

Figure 4.2 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.

4.2 Mispredictions based on Direction and Outcome

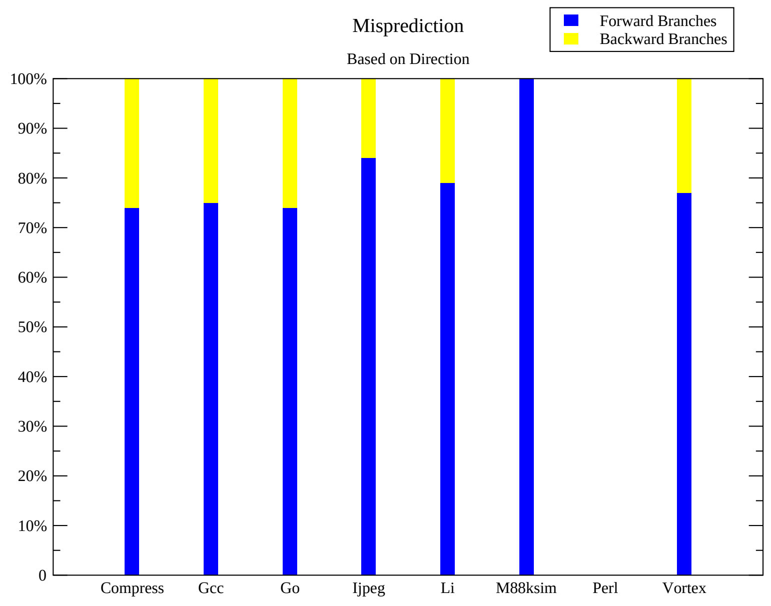


Figure 4.3: Mispredictions based on Direction

Figure 4.3 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 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 4.4 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.

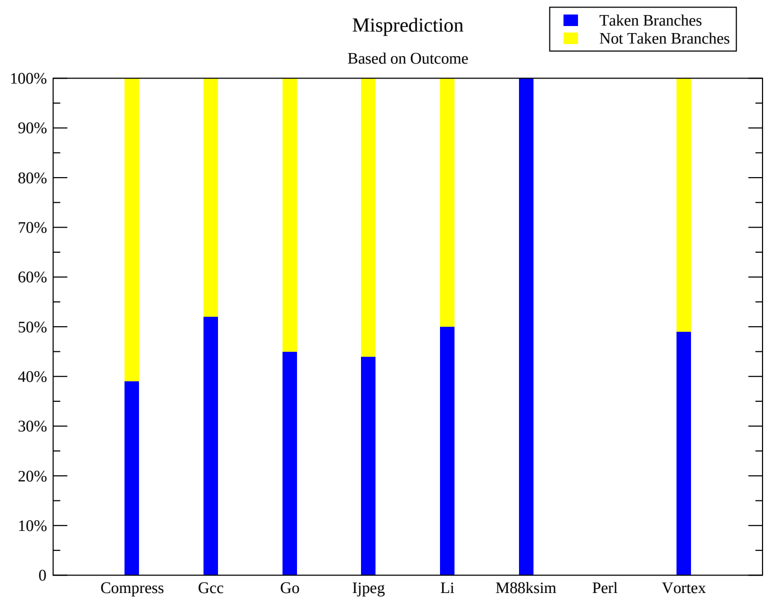


Figure 4.4: Mispredictions based on Outcome

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*.

4.3 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 4.5 shows the position of the taken targets of all direct conditional and uncon-

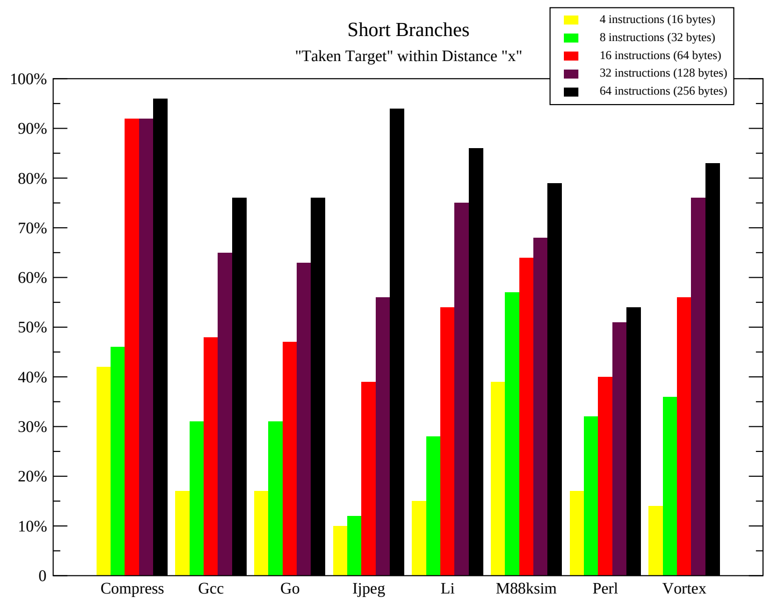


Figure 4.5: Short Branches for Different Distances

ditional branches according with a certain distance. The averages are shown on table 4.1. 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 4.1: 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.

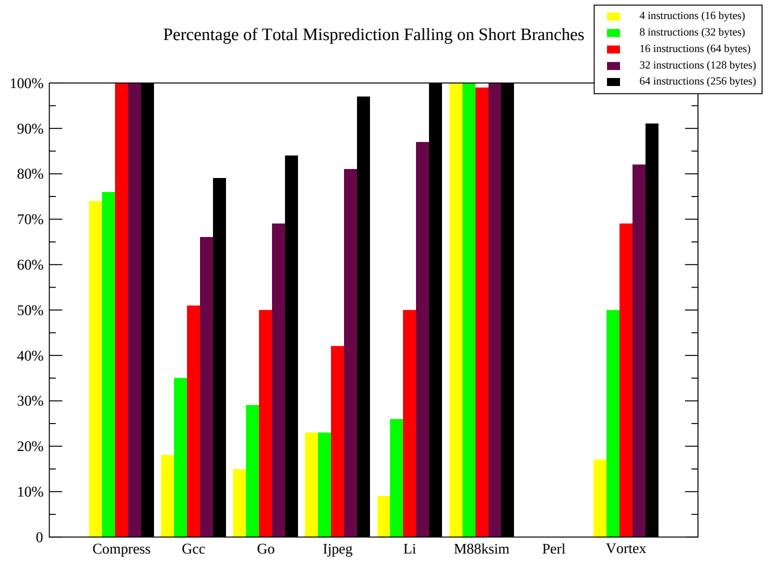


Figure 4.6: Mispredictions Falling on Short Branches

Figure 4.6 presents the percentage of mispredictions that felt within short branches. The averages are presented on table 4.1, 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.

4.4 The role of short branches

A conditional branch is a construct that may have one or two sides. An if-then type of branch has one side and a merge point. An if-then-else branch has two sides and a merge point. The merge point is a piece of code that is common to any of the possible paths of a branch.

When a branch is fetched, the processor fetches the branch itself and the adjacent path, if the branch is predicted not taken. The adjacent path starts at the instruction that follows the branch not requiring thus any redirection at the fetch engine.

When the branch is predicted taken, the processor must redirect the fetch to the target instruction. The target instruction is distant at least one instruction from the branch itself thus requiring some sort of redirection. Usually, the fetch sends the branch to the next pipeline stage and then, in the next cycle, if a cache miss does not occur, sends the target instruction if the branch is predicted taken.

Sometimes the target may be within the same cache line and therefore is fetched with the branch. To collapse the cache line and send the branch and the target in the same cycle, the processor would have to implement a complex interconnection network to remove the instructions that are sitting between the branch and the target. To simplify, the processor then sends the branch, fetches the same line once again and then sends the target.

If the branch and the target are not in the same cache line, another cache access would be required anyway. In this case the extra access does not impose a real additional penalty. However, some bandwidth is still wasted as the instructions that follow the branch have to be discarded as they are not part of the taken path.

In other words, a taken branch always imply some penalty even when the branch is predicted correctly. The penalty in this situation resides in the fact that fetch bandwidth is wasted hence reducing the number of instructions that are sent to the next pipeline stages.

The chances that the branch and the target are within the same cache line is higher when we have a short branch. This is specially interesting if we consider that both paths of the branch can be entirely fetched in one single cache access or perhaps a few cache accesses. Even if the entire branch is not within one cache line, the rest of the branch will be in the next cache line.

5 DCE - DYNAMIC CONDITIONAL EXECUTION

The Dynamic Conditional Execution model is based on the concept that multiple paths of certain branches are fetched and executed conditionally. The conditional execution is also known as dynamic predication.

A branch is conditionally executed when both paths are fetched and executed but only one path commits. The selection of a branch for dynamic conditional execution is done by the compiler that analyzes the code prior to the execution and marks those branches that are suitable for predication.

A branch can be easier predicated if it is a short branch because it will not require many fetch cycles to send the entire branch (both paths) to the execution core. Another condition that applies to the selection of a branch for predication is the type of instructions in the branch's body (section 5.3).

When a branch is marked for predication, DCE will fetch both paths and send them to the execution core. The rename will modify the instructions format to manage inter-path false dependences allowing both paths to be executed in parallel according to the availability of functional units.

When a predicated branch commits, the wrong path is selectively squashed. The scheme allows to reduce the number of predicted branches reducing thus the number of mispredictions that would affect the processor's performance.

5.1 DCE pipeline at a glance

5.2 Fetch Unit

The fetch in DCE is very similar to the fetch in a conventional superscalar machine. The instruction cache (icache) is accessed and a line is retrieved when there is a hit. If the line misses in the icache, the fetch unit waits until the line is brought in from the lower levels of the memory to continue the processing.

The line is then processed in order to look for branches that may tell the fetch engine to redirect the fetch in the next cycle. This processing is usually done by accessing branch prediction tables indexed by the PC (Program Counter) address of the instructions being fetched at the moment.

When a PC address hits in the branch prediction table, the outcome and direction are predicted based on the past history of that specific branch or based on the pattern of the branch. According to the predicted direction, an address retrieved from the branch target buffer (BTB) is used to guide the next fetch, if the direction predicted determines that the branch should be taken. When the branch is determined to be not taken, the next address is the sequential address following the PC of the last instruction fetched (PC + 1 instruction).

What distinguishes DCE from a conventional machine is that some instructions will come marked with a predication bit. The predication bit tells the fetch unit that the instruction can be predicated and is provided by the compiler or the loader (discussed in section 5.3). The predication bit is a static information that qualifies an instruction for predication.

Based on the predication bit, the fetch switches between two modes of operation: *prediction* or *predication* (figure 5.1).

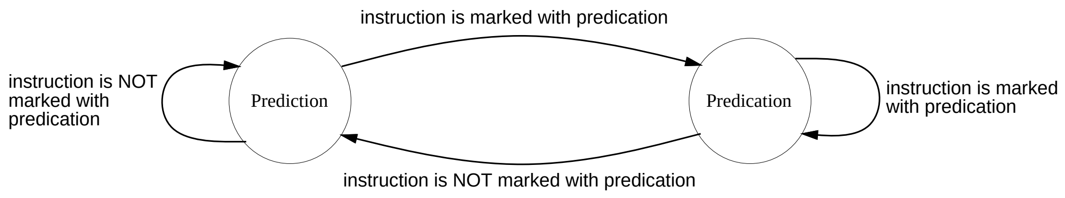

instruction is marked with predication
instruction is NOT marked with predication
Prediction
Predication
instruction is marked with predication
instruction is NOT marked with predication

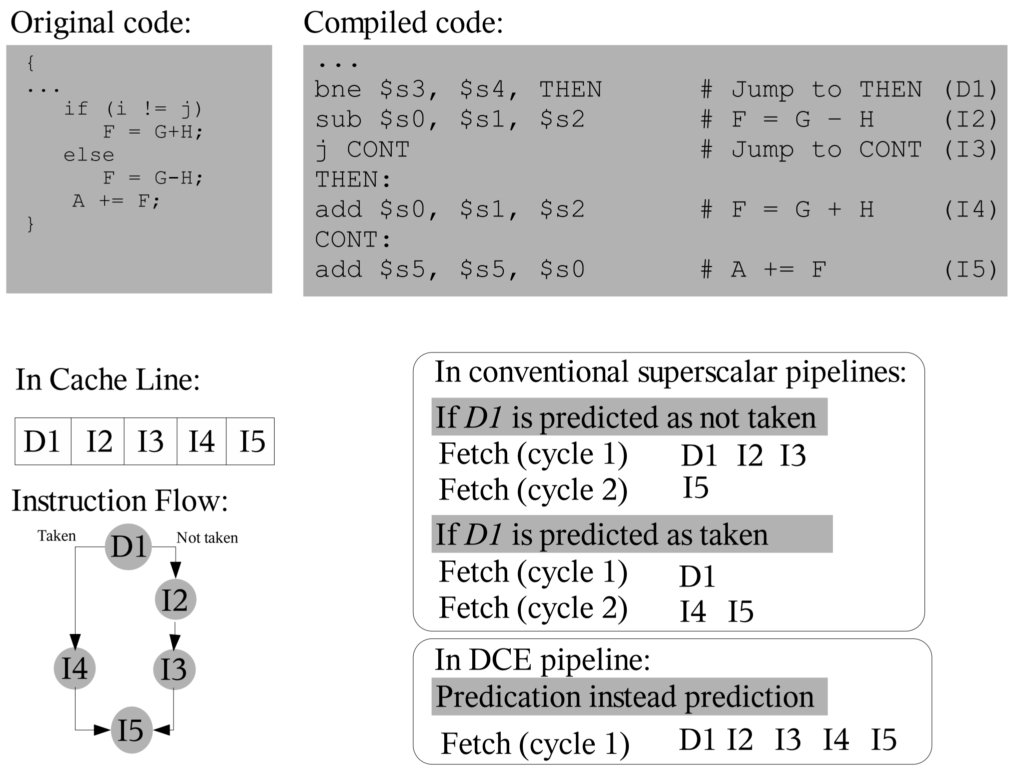
Figure 5.1: Fetch unit finite state machine

When operating in *prediction* mode, the fetch will predict any branches encountered in the instruction bundle being fetched. The prediction involves accessing the prediction tables and finding a direction and address prediction. If the instruction misses in the prediction tables, and it is a branch, the prediction will simply return the next PC (PC + 1, since the branch address was not found in the prediction table) and the fetch continues sequentially until the branch resolves and, if it is taken, the fetch redirects to the taken address after recovering from the misprediction.

When operating in *predication* mode, the fetch does not check for branches. The fetch increments the PC for every instruction fetched until it finds an instruction that is not marked for predication. In this moment it switches back to *prediction* mode and starts again checking for branches and predicting the next PC whenever a branch is encountered in the fetch bundle.

```
/*****************************************************************/
  PC = NextPC;
  for (i=0; i < fetch_width; i++) {
     if (cache_line == hit) {
        if (instruction is not marked for predication) {
           if (branch)
              NextPC = predict(branch);
           else
              NextPC = PC + 1;
           if (NextPC != (PC + 1))
              wait until next cycle;  /* branch predicted taken */
        }
        else {         /* instruction is marked for predication */
           NextPC = PC + 1;
        }
    }
    else {
       NextPC = PC;
       wait until miss resolves; /* blocking icache */
    }
  }
/*****************************************************************/
```

The code above provides a high level overview of the fetch algorithm. Notice that when in *predication* mode, the fetch will not check for branches which avoids accesses to the prediction tables. One of the benefits of DCE is that there is a lower number of accesses to the prediction tables and the overall number of predicted branches is lesser than in a conventional machine.



Original code:
{
...
if (i != j)
F = G+H;
else
F = G-H;
A += F;
}
Compiled code:
...
bne $s3, $s4, THEN # Jump to THEN (D1)
sub $s0, $s1, $s2 # F = G - H (I2)
j CONT # Jump to CONT (I3)
THEN:
add $s0, $s1, $s2 # F = G + H (I4)
CONT:
add $s5, $s5, $s0 # A += F (I5)
In Cache Line:
D1 I2 I3 I4 I5
Instruction Flow:
Taken
Not taken
D1
I2
I3
I4
I5
In conventional superscalar pipelines:
If D1 is predicted as not taken
Fetch (cycle 1) D1 I2 I3
Fetch (cycle 2) I5
If D1 is predicted as taken
Fetch (cycle 1) D1
Fetch (cycle 2) I4 I5
In DCE pipeline:
Predication instead prediction
Fetch (cycle 1) D1 I2 I3 I4 I5

Figure 5.2: Fetch comparison between DCE and a conventional superscalar machine

Figure 5.2 presents a comparison demonstrating the difference between a fetch in DCE and in a conventional superscalar machine. A conventional machine would make a prediction whenever it sees branch (*a*) coming into the fetch unit. The prediction can be either taken or not taken. When the prediction is taken, the fetch is interrupted. The fetch continues to the next instruction, at the target address (*d*) in the next cycle only.

If the prediction is not taken, still there is a taken branch (*c*) that will possible be predicted as taken, as it is an unconditional branch, and will also introduce a break in the fetch bundle. In both cases, it is required two fetch cycles to fetch all instructions down either one of the paths.

In DCE, branch (*a*) can be marked as instruction valid for predication. In this case, the fetch will send all instructions in the cache line to the next stage in one cycle without introducing any breaks. If branch (*a*) is not marked for predication, the fetch will then operate in prediction mode and the processing will be identical to the conventional machine. Whether the predication bit is set for branch (*a*) or not will depend on the preprocessing of the source code done by the compiler, or the loader, discussed in the next section.

Notice that when prediction is applied, in either case, instruction (*e*) is fetched and possibly executed. Instruction (*e*) is said to be in the convergent point, or merge point, of branch (*a*). Instructions in the merge point of a conditional branch are in the control independent (CI) path of that branch.

5.3 Compiler Support

A predicated branch has both paths executed and after its execution the incorrect path is canceled. Because, execution and squashing of the incorrect path is done dynamically, this technique is called dynamic conditional execution or dynamic predication.

Predicted branches have only one path executed at a time. If the path is correct, i.e. the prediction was correct, the entire path is committed. If the path is incorrect, then it is discarded and the fetch redirected to the correct fetch address.

The compiler analyzes the static code and looks for conditional forward branches. It then classifies the branches according to the number and type of nested branches that they may have and the distance to the taken target.

A conditional forward branch that has no nested branches may have one (if-then) or two sides (if-then-else). These branches are called simple hammocks single sided (*a*) or double sided (*b*), respectively (figure 5.3).

Conditional forward branches that have other conditional forward branches inside may be classified as follow (figure 5.4):

1. one or more nested conditional forward branches totally contained are called *pure complex hammocks (a)*

2. one or more conditional forward hammocks whose target address coincide with the join address of the outer hammock are called *multiple join complex hammocks (b)*

3. one or more conditional forward branches whose target address is beyond the join address of the outer hammock are called *multiple target complex hammocks (c)*

4. one or more conditional forward branches whose target address targets the body of the taken path are called *overlap complex hammocks (d)*

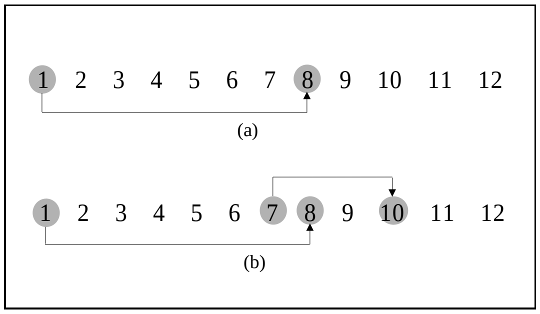


Figure 5.3: Example of simple hammocks

Other combinations of nested hammocks are also possible. The combinations showed here are the base combinations recognized by the preprocessing compiler. The compiler initially classifies a given branch into one of the classes below. At last, it looks into the final combination, that is, a combination of one or more of the classes, and evaluates if it is still a valid combination.

A hammock may not qualify for predication due to the occurrence of any of the following:

- backward branches
- indirect branches
- unconditional jumps that are NOT related to one or more conditional branches
- subroutine calls or returns
- system calls

Figures 5.3 and 5.4 presented the five different hammock classifications. The diagrams presented there are such as each number corresponds to an instruction and each arrow represents a branch to a given target, i.e. another instruction. The source of the arrow is the instruction which originated the branch (conditional or unconditional). The end of the arrow indicates the taken target instruction.

Figure 5.3 (a) presents the most basic hammock structure. It is a typical *if-then*, where a condition is investigated in instruction *1* and, depending on the result, the instruction flow continues sequentially or is redirected to instruction *8*. Note that instruction *8* is part of any flow path started in *1*, taken or not, so it is called join point. Because in this structure there are no nested branches and there is only one side (*if-then*), this category is called *One-sided Simple*.

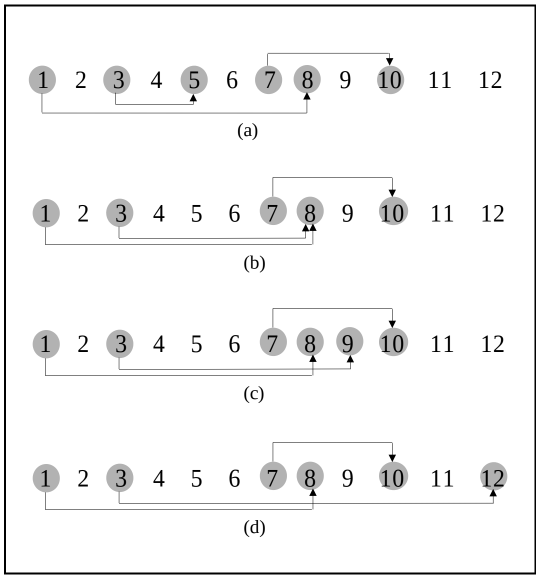


Figure 5.4: Example of complex hammocks

Figure 5.3 (b) shows a more sophisticated hammock. The example corresponds to an *if-then-else*, which has a condition evaluation in instruction *1* and a unconditional branch in a later instruction, represented by instruction *7*. This unconditional branch is responsible for the flow redirection demanded by the *else* command. It is possible to see that the unconditional branch is the instruction right before the target of *1*, i.e. instruction *8*. In this example, the join point is given at instruction *10* and the category is called *Two-sided Simple*. For instance, for a branch to fall into this category it must have the unconditional jump right before the target of the first conditional branch (branch delay slots were not considered). If this condition is not true then the branch falls off the category.

Simple hammocks are *if-then* and *if-then-else* type of constructs with no nested branches except the unconditional jump which is exclusively used to implement the *if-then-else* construct. All other hammocks are essentially different variations of these two types although including other nested hammocks. If a hammock has any nested hammocks it is then called *Complex*. It is important to mention that the classification for complex branches presented here extends the terminology and analysis once presented by (KLAUSER et al., 1998) addressing in more details the peculiarities of complex hammocks.

An example of a *Complex* pattern is showed in Figure 5.4 (a). In this case, there are nested hammocks within the outer hammock. The outer hammock is an *if-then-else* similar to the one presented earlier, where instruction *1* jumps to *8*, if the condition evaluated is true. Furthermore, there is an unconditional jump in instruction *7*, jumping to the join point, i.e. instruction *10*. Inside this hammock, instruction *3* is a simple *if-then*, like the one in the first example of this section. For this example of nested hammocks, the target of the second branch is totally contained within the most external hammock, instruction *5*. When all nested branches have their targets totally contained within the boundaries of the most external branch, that branch is called *Complex Pure*.

Figure 5.4 (b), presents an *if-then-else* hammock with multiple join points. This means that one of the sides of the most external branch (*then* or *else*) has a nested branch whose target is the same as the first, most external branch. In the example, instruction *8* is the target of two conditional branches (instructions *1* and *3*). Then, the most external branch has the same target as the most internal branch and then it is called *Complex with multiple joins*. The join point of this hammock is instruction *10* as this instruction is the first instruction common to any path starting in *1*. Observe though that branch *3* would have a join point at *8* if it was not a nested branch of *1*. When classifying complex hammocks the join point is considered to be the first instruction common to all paths starting from the outer hammock.

In other cases, nested branches may not have targets that are coincident with the target of the external branch. In this cases the target may be inside the else path while the branch is within the then path or it may be beyond the join point of the external branch, Figures 5.3 (c) and 5.4 (d).

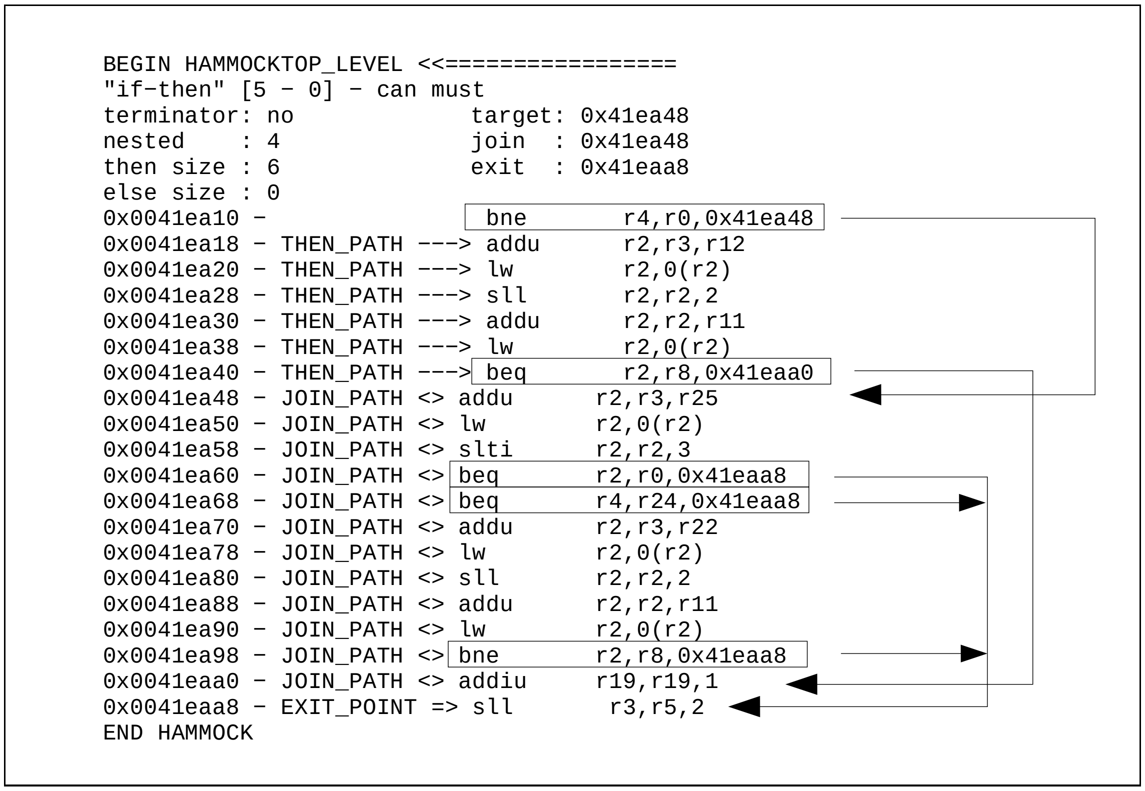
When the target of a nested branch, located within the then path, is actually in the else path of the most external branch, example (c), the two branches are overlapped and the category in each they are included is called *Complex overlapped*. The join point is still the common instruction to all paths, i.e. instruction *10*.

Example (d) shows the nested branch *3* which has a target *12* beyond the join point of the most external branch *1*. In this case the join point is instruction *12* as it is the first instruction common to any path that starts in *1*. This type of Complex branch is called *Complex with multiple targets*.

DCE was designed to handle any possible combinations of the above described categories of simple and complex hammocks.

The source preprocessing was implemented as a parser that runs before a simulation and marks groups of instruction that qualify for predication. The instructions are simply marked with one bit of information and the code is loaded into the main memory.

Although we refer to this process as compiler support, it could be implemented in



```
BEGIN HAMMOCKTOP_LEVEL <<=================
"if-then" [5 - 0] - can must
terminator: no             target: 0x41ea48
nested    : 4              join  : 0x41ea48
then size : 6              exit  : 0x41eaa8
else size : 0
0x0041ea10 -                bne       r4,r0,0x41ea48
0x0041ea18 - THEN_PATH ---> addu      r2,r3,r12
0x0041ea20 - THEN_PATH ---> lw        r2,0(r2)
0x0041ea28 - THEN_PATH ---> sll       r2,r2,2
0x0041ea30 - THEN_PATH ---> addu      r2,r2,r11
0x0041ea38 - THEN_PATH ---> lw        r2,0(r2)
0x0041ea40 - THEN_PATH ---> beq       r2,r8,0x41eaa0
0x0041ea48 - JOIN_PATH <> addu      r2,r3,r25
0x0041ea50 - JOIN_PATH <> lw        r2,0(r2)
0x0041ea58 - JOIN_PATH <> slti      r2,r2,3
0x0041ea60 - JOIN_PATH <> beq       r2,r0,0x41eaa8
0x0041ea68 - JOIN_PATH <> beq       r4,r24,0x41eaa8
0x0041ea70 - JOIN_PATH <> addu      r2,r3,r22
0x0041ea78 - JOIN_PATH <> lw        r2,0(r2)
0x0041ea80 - JOIN_PATH <> sll       r2,r2,2
0x0041ea88 - JOIN_PATH <> addu      r2,r2,r11
0x0041ea90 - JOIN_PATH <> lw        r2,0(r2)
0x0041ea98 - JOIN_PATH <> bne       r2,r8,0x41eaa8
0x0041eaa0 - JOIN_PATH <> addiu     r19,r19,1
0x0041eaa8 - EXIT_POINT => sll       r3,r5,2
END HAMMOCK
```

Figure 5.5: Hammock debug output

reality as a loader. Instead of augmenting the original binary code, this process could be performed during the load phase and thus would not require recompilation.

Figure 5.5 shows a segment of the output produced by the DCE simulator with the *-hammock:debug* option turned on. The segment shows a valid hammock being marked by the parser. Notice that there are 4 nested hammocks.

The outer hammock (*0x0041ea10*) is a if-then type of hammock which has one nested hammock (*0x0041ea40*). The *JOIN_PATH* is in fact the merge point of the first hammock. As the first hammock is an if-then, the merge point starts at the target instruction.

The nested hammock (*0x0041ea40*) starts a new series of hammocks that will then form the whole group. Because the second hammock starts in the body of the first one, the parser needs to analyze its body as well. In looking into the body of the nested hammock, the parser encounters other three hammocks that have coincidently the same target. This is a very complex construct derived from optimizations that the original compiler applied to the source code. DCE can handle this combination as there are no invalid hammocks nested in any of the hammocks. The group starting at *0x0041ea10* and finishing at *0x0041eaa8* is then marked as valid for predication.

When this segment of instruction is being fetched, the fetch unit will detect the predication bit on and will switch to predication mode. It will continue in predication mode until it finds the instruction that follows *0x0041eaa8* which will have the predication bit off. Then it switches back to prediction mode and the process repeats itself until the execution finishes.

Notice that even though the join point of the first branch is *0x0041ea48* the real exit point is at *0x0041eaa8* as the parser needs also to consider the nested hammocks in order to evaluate and qualify the entire segment of code. The parser decodes and interprets the simplescalar pisa binary (*.ss*).

DCE will look to all instructions between these two points as if they were regular instructions and it will not predict the branches marked for predication.

The parser has the ability to check the overall distance between the start point and the exit point and qualify a hammock based on that distance too. Furthermore, the parser can also be configured to accept only certain types of hammock or all types of hammocks according to the classification presented previously. All these features can be configured by passing arguments to the simulator through a configuration file 6.

5.4 Register Renaming

The register renaming is perhaps the most important stage in the DCE architecture. Through a special register renaming technique derived from the MULFLUX architecture (CHAVES FILHO et al., 1999), DCE is capable of keeping track of several different contexts. Each in flight path has its own context which allows to handle intra-dependencies correctly and avoid inter-path false dependencies.

The renaming process can be divided into two main steps. The first step is to decode and tag instructions according to the path to which they belong and generate new tagids (target tagids) if the instructions spawn new paths. Second, is to rename the logical registers to its corresponding physical registers. The second step will be referred to as register remapping.

Two minor architectural changes are required to implement this scheme. First, a table called TTB, that stands for *Target Tagid Buffer*, is used to store outstanding tagids produced by instructions that spawn new paths and second, multiple mapping tables one per active path rather than one mapping table which would be the normal for a conventional machine.

The size of the TTB and the number of mapping tables is directly related to the number of outstanding paths supported by the architecture. If the machine supports up to 16 paths, it will need then 16 mapping tables and a TTB with 16 entries, and so on.

Although there are multiple mapping tables, there is only one physical register file that is shared by all instructions.

5.4.1 Instruction Tagging

Each path spawned has one unique tagid and all instructions belonging to a path are tagged with the path's tagid.

In DCE, an instruction fetched may create multiple replicas. It is at the tagging phase that multiple paths are in fact spawned. When a conditional branch is marked as predicated, the fetch does not make a prediction regarding its outcome. Instead, it keeps fetching instructions down the sequential stream eventually fetching the instructions of both paths of the branch as it reaches the exit point.

It will also fetch instructions from all paths of any nested branches as well. Assuming that the initial branch does qualify for predication, all nested branches will be completely contained within the boundaries determined by the start point (the initial branch) and the exit point (the first instruction common to all possible paths that start from the initial branch).

At any moment, there is at least one active tagid in the machine. As the fetch stage, the rename also operates in two modes: prediction or predication. When operating in prediction mode, it acts as if there were only one active path since a prediction has been made in the fetch stage for any branches. The tagging in this case just decodes the instruction and applies to it the active tagid(s) before sending it to the next step.

However, even when in prediction mode, the rename might need to apply several tagids to one logical instruction. This happens when there are unresolved predicated branches still executing in the machine.

In such cases, the rename works the same way as described, except that it replicates every instruction received from the fetch by the number of active tagids. For example, if instruction *a* is being tagged and there are 3 active tagids at the time, the tagging mechanism will buffer 3 replicas of instruction *a*, say, *a1*, *a2* and *a3*. Each replica belongs to a different path.

Eventually when all previous predicated branches have resolved, there will be only one replica alive and all the other replicas will be squashed. The replica that reaches the top of the reorder buffer and has a its valid bit set, is allowed to retire (see 5.8 for more details).

5.4.1.1 Tagid generation

According to the compiler, the first instruction predicated is always a conditional forward branch. When a predicated branch is tagged, the tagging mechanism will figure out the tagid of its target instruction.

The target tagid generation is very simple. The *tag_depth_level* determines the bit position in the tagid that indicates if the path is the taken or not taken path of a previous branch. Not taken paths are tagged with 0's and taken paths are tagged with 1's. The tag is placed onto the corresponding bit of the tagid for the depth of the path being tagged (Figure 5.6).

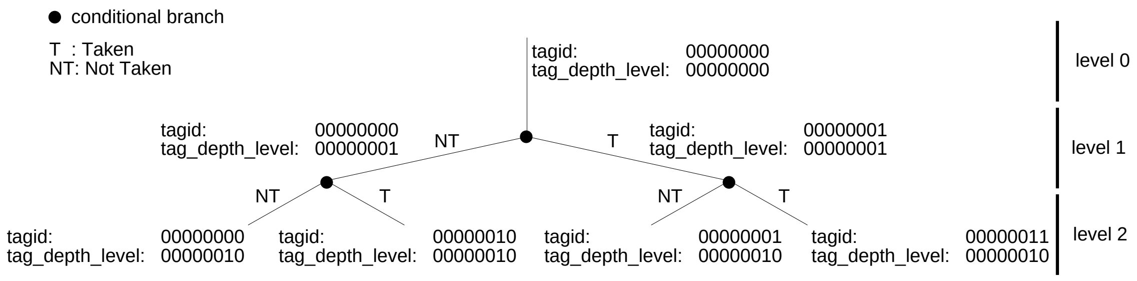


Figure 5.6: Tagid generation and use

Notice that all paths generated from the not taken path of the initial branch have the least significant bit 0 and all paths generated from the taken path of the initial branch have the least significant bit 1.

A new path created always carry the tagid of the branch that generates the path combined with the new tag (taken-1 or not taken-0) placed in the depth level corresponding to the path. Appying this technique becomes easy to differentiate the taken paths from the not taken paths.

5.4.1.2 TTB

The TTB stores the target tagid of predicated branches. When a predicated branch is tagged, the tagging mechanism will generate the target tagid by applying the technique described previously.

The new tagid will only be applied to the instructions in the taken path of the branch. As the fetch follows a sequential order and the tagging is done in order, the not taken or follow through path comes right after the branch itself. The target tagid needs to be stored temporarily.

The renaming detects that the taken path of branch has started when the address of the instruction being tagged matches an address stored in the TTB. As we saw previously (example 5.5), an instruction might be the target of several branches and thus it might cause more than one hit in the TTB when the lookup is performed..

5.4.1.3 *Tagging predicated branches*

When a branch is tagged, the target tagid and the target address are stored in the TTB together with the tag_depth_level. For every instruction tagged, the tagging mechanism must lookup the TTB first to find out if the instruction's address matches any of the entries stored.

If a matches occurs, the entry (or entries if more than one match the address) are removed from the TTB and the tagids are added to the list of active tagids. From that moment onward, all instructions are tagged with the tagids that became active.

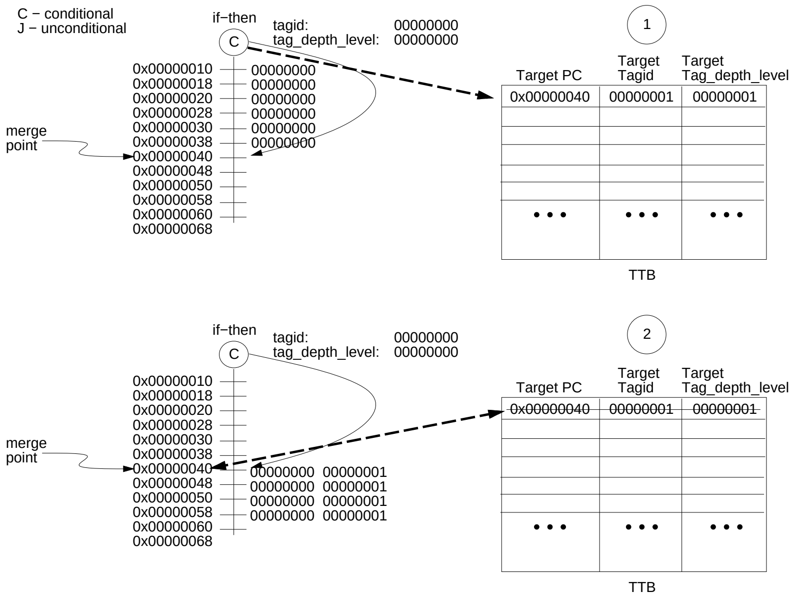


Figure 5.7: Tagging an if-then predicated branch

Figure 5.7 presents an example of a if-then segment of code being tagged. When the branch is tagged, the target address, the tagid and the tag_depth_level are stored in the TTB (1).

Later, when the target instruction (*0x00000040*) is tagged (2), its address matches the first entry in the TTB. The entry is removed and that tagid becomes active too. Notice that the instruction at the merge point is then tagged with both the current tagid [*00000000*] and the new tagid [*00000001*]. At this moment the tagging mechanisms introduces two replicas of this instruction, one for each active path. The replica tagged with [*00000000*] belongs to the not taken path and replica [*00000001*] belongs to the taken path of the branch.

Unconditional branches are only allowed when they are part of an if-then-else as shown on Figure 5.8 and 5.9. When an unconditional branch is tagged, all active tagids at the moment are stored in the TTB and the target address of the unconditional branch is stored with every entry. These tagids become inactive until the target address of a previ-

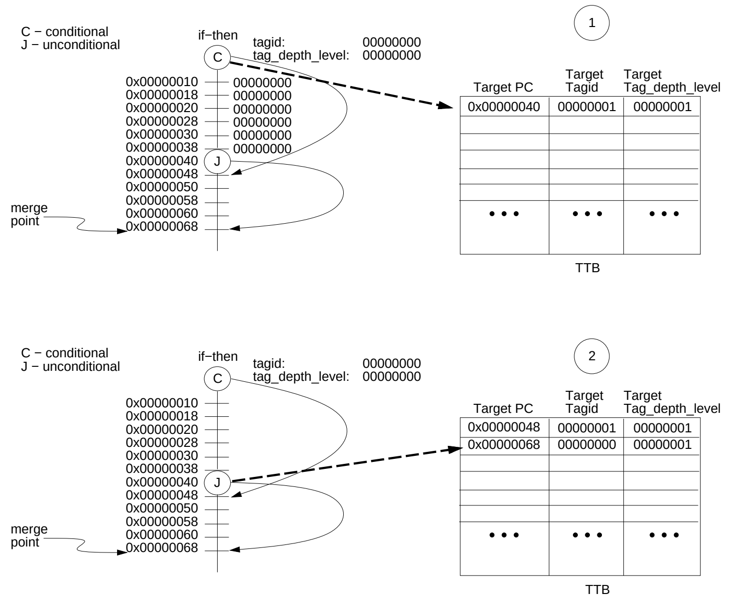


Figure 5.8: Tagging an if-then-else predicated branch (1)

ous branch hits in the TTB. When the target is reached, all tagids will be restored and will become active again.

Figure 5.8 presents the not-taken path of an if-then-else construct being tagged. Instructions between the branch (*C*) and the unconditional jump, that skips the taken path, are tagged with the current tagids. The target tagid is stored in the TTB just as it is done for an if-then branch.

When the unconditional jump is reached, all active tagids are stored in the TTB and become inactive. No tagids remain active after an unconditional branch is tagged. It is guaranteed though that the next instruction, following the unconditional branch, will match at least one entry in the TTB as the instruction must be at the target address of a previous predicated branch.

This assertion is made by the compiler when selecting instructions for predication. An unconditional branch is only allowed if it is part of an if-then-else construct. In such cases, the unconditional branch is always placed before the target of the initial branch so that it indicates the flow to skip the taken path in case the conditional branch is not taken.

When the instruction following the unconditional branch is tagged, it will always match one or more entries in the TTB. The active tagid list will then be updated with the new tagid(s). At least one tagid will always become valid.

Figure 5.9 presents the taken path of the same if-then-else predicated branch being tagged. The first instruction after the unconditional jump (*0x00000048*) matches the first entry in the TTB as it is at the target address of the initial branch.

Tagid [*0x00000001*] (3) becomes active and it is used to tag all instructions in the taken path. When the instruction at (*0x00000068*) is tagged, it will match the second entry in the TTB and then tagid [*0x00000000*] will become active again.

All instructions starting at address (*0x00000068*) will then be tagged with both tagids [*0x00000001*] and [*0x00000000*] (4). As there are two active tagids, DCE will insert two replicas of each instruction, one with each one of the active tagids.

The algorithm described above generates replicas of instructions at the merge point of predicated branches. The instruction replication allows the machine to maintain correct data dependencies and avoid inter-path false dependencies.

As each path produces different data sometimes to the same logical registers, it is necessary to guarantee that instructions that read the data will read the data produced in the path to which they belong.

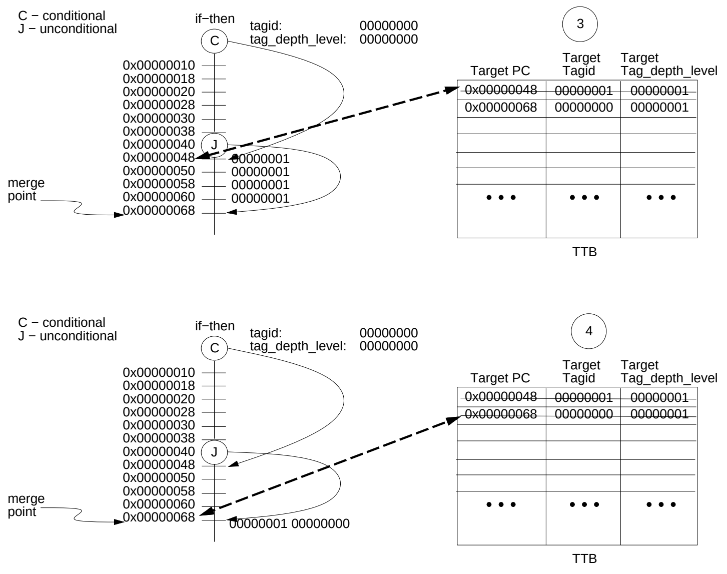


Figure 5.9: Tagging an if-then-else predicated branch (2)

In some cases, as it will be presented later, it is possible to avoid the replication when the instruction does not read data produced in one of the paths. In this case, if the instruction reads data that was produced by a non-replicated instruction that lies logically before the predicated branch, only one instance of the instruction is inserted and marked as CIDI (Control Independent Data Independent). This optimization allows to reduce part of the overhead generated by the replication of logical instructions.

5.4.1.4 Misfetch Recovery

It is possible that the fetch engine retrieves a direction prediction (taken or not taken) from the prediction tables but it is unable to find an address in the BTB or it finds an address that was stored by a different branch. Since the BTB is not infinite, a conflict may cause a branch to access an entry in the BTB that was written by a previous different branch.

When this happens and the direction predicted is taken, the fetch is unable to continue fetching instructions when it cannot retrieve an address or it is possible that the fetch will continue through a wrong address.

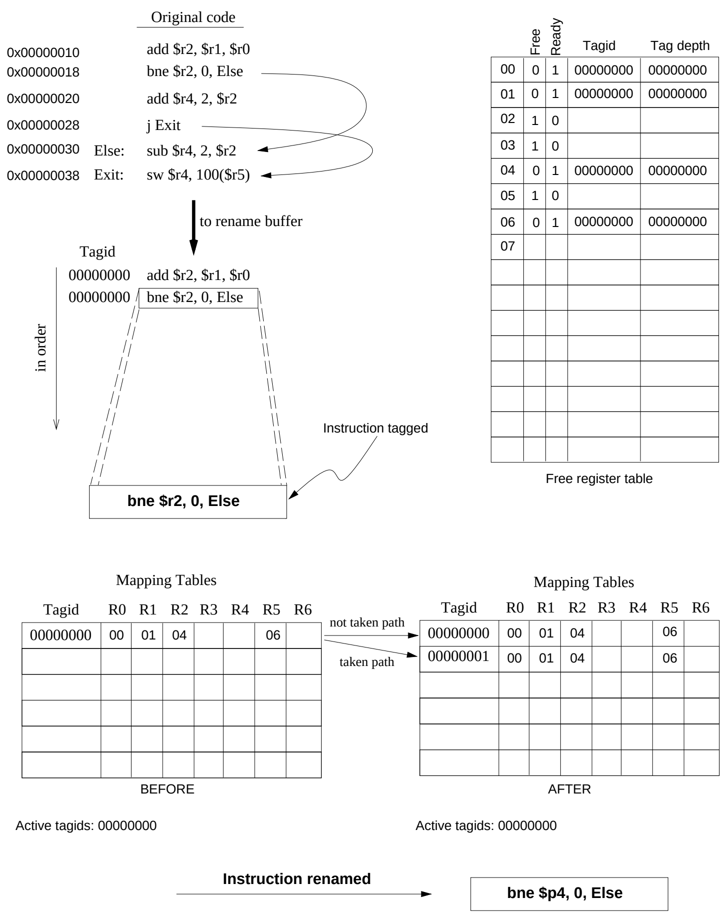
Since the direction is predicted, when the instruction is decoded it is possible to detect these situations by comparing the address decoded (if it is a direct branch) with the address used in the prediction. The rename does perform this comparison and redirects the fetch to the correct address. This will cause a misfetch penalty of a couple of cycles.

If the direction predicted is correct, this penalty still impacts the execution but the number of cycles lost is the difference between the time the instruction in the target was fetched and the time the misfetch was detected at the rename stage. If the prediction is incorrect, the misfetch penalty is overlapped with the misprediction penalty and does not add to the later. There is no misfetch penalty for predicated branches since a predicated branch does not require any redirection of the fetch.

5.4.1.5 *Mapping table allocation*

When a predicated conditional branch is tagged, a sequence of events take place in the rename. First, a new mapping table is allocated for the taken path of the branch being tagged. The contents of the current mapping table (assigned to the branch itself) is copied to the new mapping table (Figure 5.10).

Then, the tag_depth_level associated with each of the current active tagids is shifted 1 bit to the left representing that the instructions that will be tagged next are one level deeper with relation to branch that was tagged. The least significant bit of the tagid is updated with the information about the path associated to the tagid (taken-1; not taken-0). The tag_depth_level is used to generate the target tagid of future branches.



```
0x00000010          add $r2, $r1, $r0
0x00000018          bne $r2, 0, Else
0x00000020          add $r4, 2, $r2
0x00000028          j Exit
0x00000030  Else:   sub $r4, 2, $r2
0x00000038  Exit:   sw $r4, 100($r5)
```

```
Tagid
00000000   add $r2, $r1, $r0
00000000   bne $r2, 0, Else
```

	Free	Ready	Tagid	Tag depth
00	0	1	00000000	00000000
01	0	1	00000000	00000000
02	1	0		
03	1	0		
04	0	1	00000000	00000000
05	1	0		
06	0	1	00000000	00000000
07				

BEFORE:

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04			06	

AFTER:

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04			06	
00000001	00	01	04			06	

Figure 5.10: Mapping table allocation when a predicated branch is tagged

It is important to mention that the not taken path uses the same tagid and thus the same mapping table allocated for the initial branch. The new mapping table will be used later, when the taken path is remapped. Therefore, every time a new predicated branch is tagged only one new mapping table needs to be allocated.

The mapping table must be copied before any other instruction updates the contents of the initial table. By copying a mapping table, we guarantee that the context and all dependencies will be the same for the two paths that will be renamed later. From that point on, each path updates its own mapping table according to the rename of its instructions.

On Figure 5.10 we assumed that the branch at (*0x00000010*) was selected for predication and that it is being tagged. When the branch is tagged, the mapping table used in the current path, tagid [*0x00000000*], provides the contents to be copied into the new mapping table. The new mapping table is created to rename the instructions in the taken path of the branch.

The new mapping table is identified by the tagid of the target path of the branch being tagged, [*0x00000001*]. The new mapping table contains the very same information stored in the current mapping table so that it reflects exactly the same context.

The example presented here will be used in the next section to explain the register remapping procedures.

5.4.2 Register Remapping

We are assuming that all instructions have been tagged according to the tagging mechanism described previously and that the first instruction (*0x00000010*) and the branch itself have been (*0x00000018*) tagged and renamed. We also assume that the free register table holds a consistent state.

The second step of renaming is called register remapping. After instructions have been tagged, the rename will remap all source logical registers to their equivalent physical registers and change the opcode of the instruction to carry the physical id of the registers to be used during execution, instead of the logical value.

The only time an instruction requires access to the mapping table is when it is remapped. Any changes made to the mapping tables by any instruction will only affect younger instructions.

While remapping the instruction, the remapping mechanism will access the mapping table allocated for the path which the instruction belongs to and retrieve the physical register mapped to the corresponding logical registers of the instruction. It will also allocate a new physical register for the result produced by the instruction when it executes, if any. The entry in the mapping table corresponding to the logical destination register will be updated with the new physical destination register.

If the logical destination register was already allocated by a previous instruction, the entry will be updated with the new physical register and the previous destination register will be stored with the instruction in the rename buffer, for use at the commit stage.

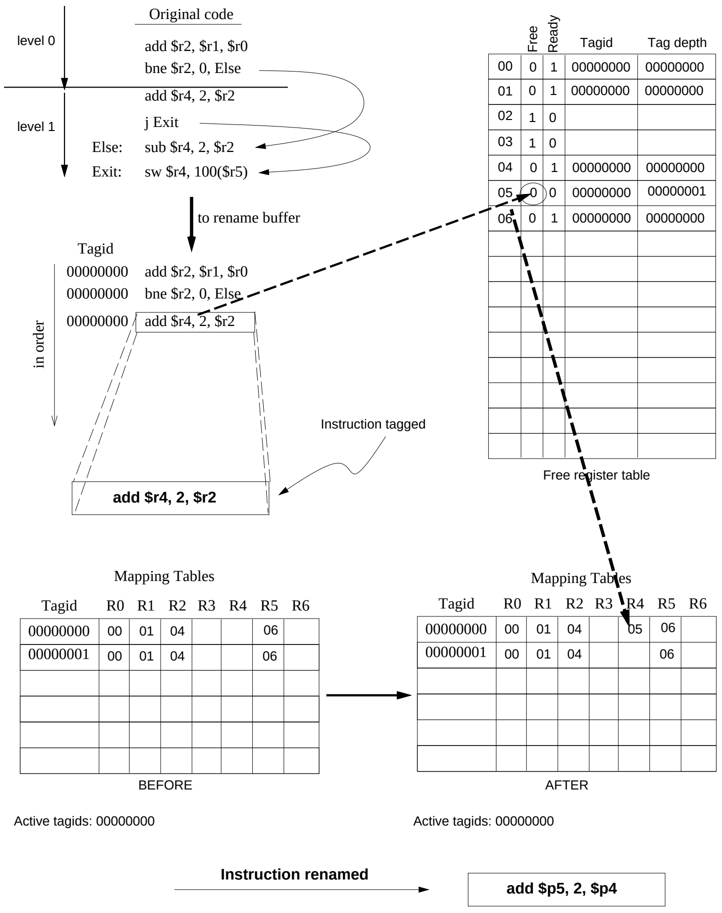
The instruction's entry in the rename buffer is updated with the following information:

- physical source registers retrieved from the path's mapping table
- physical destination register allocated for the instruction
- previous destination register, if there was one allocated by a previous instruction writing to the same logical register

The previous destination register is released when the instruction commits. If the instruction does not commit, that is, it is squashed because it is part of a wrong path, the previous destination register is simply ignored as it may still be valid in some other path.

The renaming also maintains a table of free physical registers. The free register table keeps the status of all physical registers in the machine. The status of a physical register comprises the following information:

- Free bit: indicates whether the physical register is allocated or not
- Ready bit: indicates that the instruction that allocated the register has written the result to the register. The register is ready to be read by any consumer instruction.



	Free	Ready	Tagid	Tag depth
00	0	1	00000000	00000000
01	0	1	00000000	00000000
02	1	0		
03	1	0		
04	0	1	00000000	00000000
05	0	0	00000000	00000001
06	0	1	00000000	00000000

BEFORE

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04			06	
00000001	00	01	04			06	

AFTER

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04		05	06	
00000001	00	01	04			06	

Figure 5.11: Instruction renaming down the not taken path

- tagid and tag_detph_level of the instruction that allocated the register

Figure 5.11 presents the remapping of the first instruction in the not taken path of branch (*0x00000018*). As we tag new predicated branches, the depth level is increased. Note that the instruction being remapped is now at level 1, rather than 0.

The instruction at (*0x00000020*) requires a destination register to store the result of the addition. The renaming of this instruction is based on the contents of the mapping table identified by tagid [*0x00000000*] as this is the tagid assigned to the instruction in the previous stage.

Firstly, the remapping will retrieve the source registers from the mapping table and update the instruction word at the moment it is inserted in the rename buffers. The instruction reads the contents of logical register *$r2*, adds 2 and stores the result into the logical *$r4*.

By accessing the mapping table, the remapping determines that currently register *$r2* is mapped to the physical register *$p4*. The second source operand is an immediate so it does not require access to the mapping tables. The value is passed on to the rename buffer.

Secondly, the remapping will determine which physical register will hold the result produced by this instruction, when it executes, by picking a free register from the free register table. In the example, register *$p5* is picked and marked as allocated (free bit set to 0) and not ready (ready bit set to zero). The tagid and tag_depth_level of the physical register are updated with the tagid and tag_depth_level of the instruction.

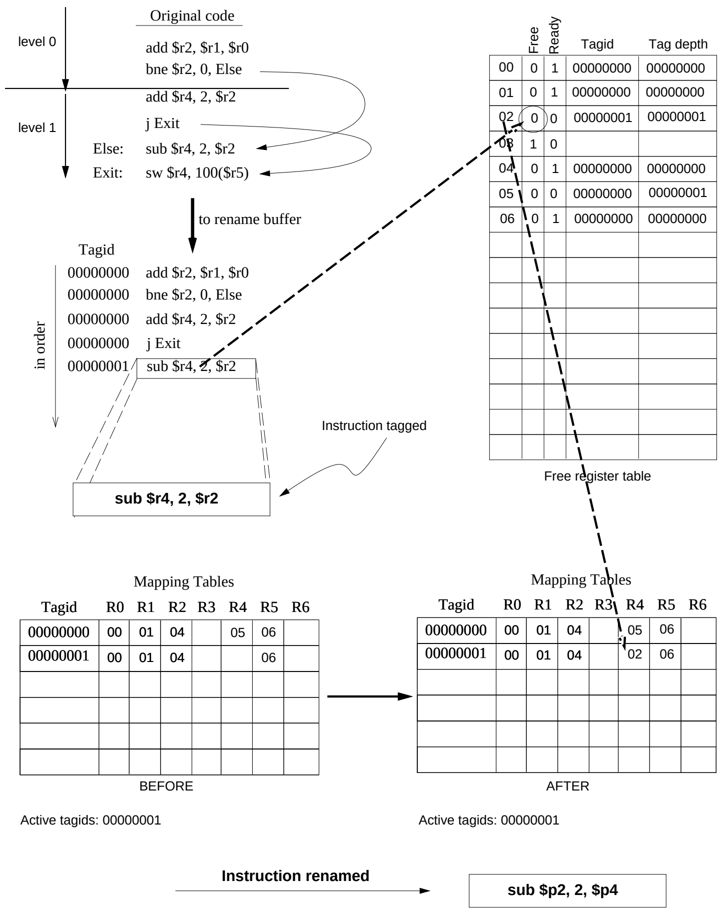
As part of this process, the remapping mechanism needs to update the mapping table to reflect the remapping just performed. The entry for logical register *$r4*, on mapping table with tagid [*0x00000000*], is updated with the physical register id allocated for the instruction. In this case, entry *$r4* will be updated with the value *5* which points to the physical register *$p5*.

As explained previously, a mapping table has one entry for each logical register but it does not require one entry for each physical register. In general, machines that use similar rename techniques will have more physical registers than logical registers.

Note also that after remapping, the two mapping tables allocated will have different contents. As the remapping of this instruction only affects the not taken path, only mapping table [*0x00000000*] is updated.

The renamed instruction inserted in the rename buffer now have the ids of the physical registers to be used in the execution. No information about the logical registers is necessary.

The jump (*J Exit*) is inserted in the pipe but it is executed as a NOP. No control transfer whatsoever is triggered by an unconditional jump at the end of the not taken path of a predicated branch.



	Free	Ready	Tagid	Tag depth
00	0	1	00000000	00000000
01	0	1	00000000	00000000
02	0	0	00000001	00000001
03	1	0		
04	0	1	00000000	00000000
05	0	0	00000000	00000001
06	0	1	00000000	00000000

BEFORE:

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04		05	06	
00000001	00	01	04			06	

AFTER:

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04		05	06	
00000001	00	01	04		02	06	

Figure 5.12: Instruction renaming down the taken path

Figure 5.12 presents the remapping of instruction (*0x00000030*) which is the first instruction in the taken path of branch (*0x000000018*).

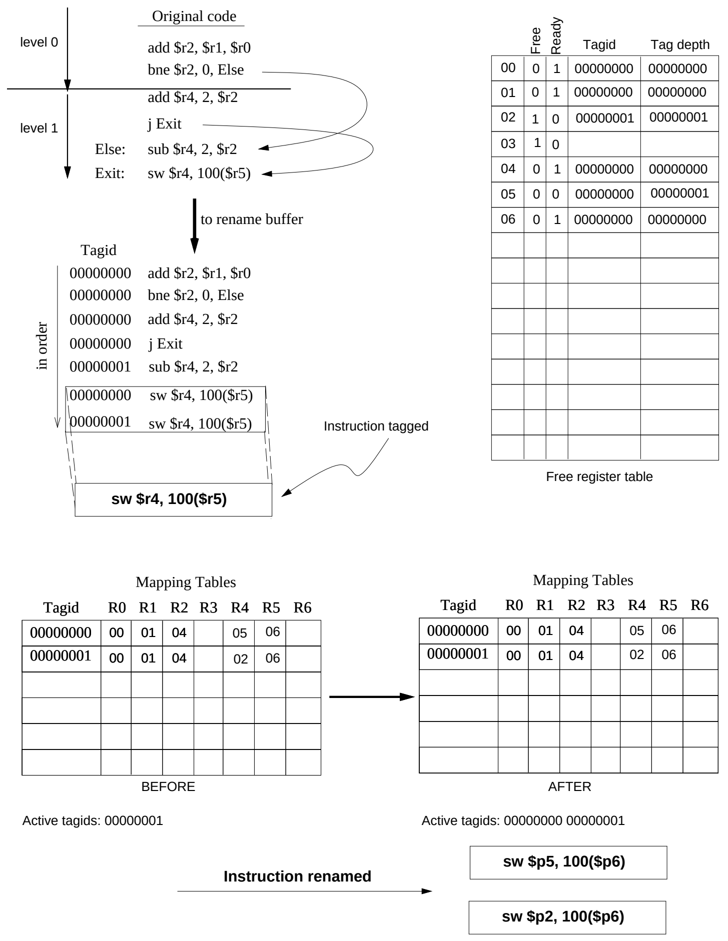
The instruction was tagged with tagid [*0x00000001* so mapping table [*0x00000001* will be used for remapping the source and destination registers of this instruction.

In the example, the instruction has the same logical source and logical destination registers of the previous instruction. This was intentionally modeled for illustrative purposes.

The source register *$r2* is mapped to physical register *$p4* just as it was for the previous instruction in the not taken path. Note that register *$p4* was allocated by the first instruction in the code sequence presented. The first instruction is logically before the branch so its result must be visible to all instructions that follow it.

The destination register though is has to be mapped to a different physical register to avoid an inter-path false dependency. The process to pick a free register is the same. Physical register *$p2* is now picked and updated with the respective tagid and tag_depth_level information of the instruction. The register is also marked as allocated (free bit set to zero) and not ready (ready bit also set to zero).

Finally, mapping table [*0x000000001] is updated with the new physical destination register. In this case, entry* $r4 *points to physical register* $p2 *and the instruction is later inserted in the rename buffer.*



```
Original code
level 0   add $r2, $r1, $r0
          bne $r2, 0, Else
          add $r4, 2, $r2
level 1   j Exit
   Else:  sub $r4, 2, $r2
   Exit:  sw $r4, 100($r5)
```

```
Tagid
00000000   add $r2, $r1, $r0
00000000   bne $r2, 0, Else
00000000   add $r4, 2, $r2
00000000   j Exit
00000001   sub $r4, 2, $r2
00000000   sw $r4, 100($r5)
00000001   sw $r4, 100($r5)
```

	Free	Ready	Tagid	Tag depth
00	0	1	00000000	00000000
01	0	1	00000000	00000000
02	1	0	00000001	00000001
03	1	0		
04	0	1	00000000	00000000
05	0	0	00000000	00000001
06	0	1	00000000	00000000

Mapping Tables (BEFORE)

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04		05	06	
00000001	00	01	04		02	06	

Mapping Tables (AFTER)

Tagid	R0	R1	R2	R3	R4	R5	R6
00000000	00	01	04		05	06	
00000001	00	01	04		02	06	

Figure 5.13: Instruction renaming at the merge point

*When instruction (*0x00000038*) was tagged, both tagids [*0x00000000*] and [*0x000000001*] were used and two copies of the instruction were inserted in the rename buffer. This was done because the instruction lays in the merge point of the predicated branch and reads a logical register that is produced in both paths of the branch (*$r4*).*

*When the copy tagged with tagid [*0x00000000] is remaped, register *$r4* is remapped to the corresponding physical register *$p5*, according to the information in the mapping table [*0x00000000*]. On the other hand, when the copy tagged with [*0x000000001*] is remapped, the logical register *$r4* is remapped to physical register *$p2* so that both instructions can be executed as soon as the source registers are ready and it does not need to wait until the branch resolves nor it creates any inconsistency with relation to the data.

In the case of this instruction in special, because it is a store instruction, the result will only be committed to the memory when the instruction actually commits so the data produced by the squashed path will not be recorded in the memory. The data though can be forwarded to future loads to the same address to avoid stalls in the issued. The data forwarding process is based on the address of the load instruction and the tagid of the store instruction so a store only forwards a data to a load of equivalent tagid.

5.4.2.1 Recovery Checkpoint

The mapping tables are updated during the second phase of the rename, called remapping. When a conditional or indirect unconditional branch is remapped a state checkpoint is saved for recovering if the branch is mispredicted.

The checkpoint saves the contents of the branch's mapping table. If the branch is mispredicted, all mapping tables are flushed and the saved mapping tables becomes the only active mapping table (see 5.8 for more details).

5.4.3 Rename Stall conditions

Below the conditions that may stall the rename, tagging and remapping are listed:

1. the rename buffer is full (tagging)

2. the available (remaining) rename bandwidth in a given cycle does not accommodate all replicas of a predicated branch in this cycle: the tagging stage will check whether there are enough bandwidth, TTB entries and mapping tables to tag all replicas of a given predicated branch or not. If the bandwidth available is not enough to tag all replicas, the tagging will get stalled until enough resources become available. This guarantees that all tags are correctly created and that this process is not going to be effected by an older predicated branch resolving which will eventually update all active tags.

3. the last instruction tagged was a system call: the tagging will tag all replicas of a system call or the only system call if there is only one path active but will stop tagging further instructions until the system call is dispatched and executed. This guarantees that system calls will be only executed non-speculatively and thus it is not necessary to have multiple replicas of instructions that come after the system call.

4. there is not enough TTB entries to store all active tagids when an unconditional branch is going to be tagged: the tagging will stop before am unconditional branch is tagged if there are not enough entries in the TTB to store all active tagids at that moment.

5. there is no physical register available to remap an instruction: the remapping procedure cannot proceed renaming instructions when there is no a physical register to allocate for an instruction. If the instruction produces no output data, the remapping is done for the sources . In this case the stall is not required.

6. the commit is waiting to commit a predicated branch: a simple handshaking mechanism synchronizes the operation of the rename (tagging) and commit stages, in certain cases. When a predicated branch commits, all active tagids need to be updated in order to invalidate the wrong paths. Because of this, a predicated branch cannot commit during the tagging of new predicated branches.

   Every time the tagging will start tagging of a predicated branch it signals to the commit that this operations is going to take place. The commit, on the other hand, needs to check whether the renaming is creating new tagids before it commits a predicated branch. If the renaming is creating tagids, the commit will wait until the predicated branch has been renamed. It signals the rename though that it is waiting to commit a predicated branch. The rename will not tag new predicated branches until the commit has completed the retirement of the oldest predicated branch.

5.5 Dispatch

The dispatch of DCE is pretty much the same as in the reference machine. Instructions are tagged and renamed in the previous stages (rename) and then enter the reorder buffer in order at the dispatch stage. The dispatch moves instructions from the rename buffer into the reorder buffer (ROB) and allocates entries in the load store queue (LSQ) for the loads and stores dispatched.

The only resource that may stall the dispatch is the availability of entries in the ROB or LSQ. A load or store instruction allocates an entry in the ROB for the instruction the calculates the load/store address and an entry in the LSQ for the load/store itself. The dispatch may also be stalled though when a system call instruction is to be dispatched. The dispatch will wait until the ROB have drained to dispatch the system call to guarantee that the instruction will not be executed speculatively.

5.6 Issue

The issue is slightly different in DCE. The only additional feature is that loads must be checked against pending stores to the same path or ancestor paths. Any pending stores will propagate the values to the store being issued even if the store is partial.

Partial stores will forward the partial data that will be combined with other partial or full stores in the order they reside in the LSQ. If there is one store which is in the same path or in an ancestor path and its address or data is unknown then any subsequent load, in a path that could be effected by the store, cannot be issued. When both data and addresses of all older ancestor stores are known then the load can be issued.

Loads can be issued speculatively. If a load obtains a full data from the LSQ, it does not require to access the memory and thus it can complete immediately. If the load obtains a partial data from a previous ancestor store, it still requires an access to the memory to retrieve the remaining bytes. The partial data, obtained from the LSQ, overlaps the respective portion of the load data and the final data is written to the destination register of the load instruction.

To determine whether a tagid is a true ancestor of another tagid, one must compare all bits of both tagids up to the tag_depth_level of the supposedly ancestor. If all bits are the same, they have the same ’root’ and therefore the first is an ancestor of the later. If both tagids are at the same tag_depth_level, there is no ancestor between the two. But, in the case of the load/store data forwarding scheme, if the store has the same tagid and tag_depth_level of the load, then they are part of the very same path and thus the store data will change or provide the load result, depending on whether the store is partial or total, respectively.

For other instructions, the issue mechanism will check whether all source operands for an instruction are ready and if so the instruction will be issued if there is a functional unit available of the type of the instruction. The functional unit allocated will remain allocated for the instruction until the instruction completes the execution according to the its latency.

The issue is out-of-order and thus any instruction in the ROB which has all source operands available at a given time and has a potential functional unit available can be issued. The issue does not follow any order when determining what instructions can be issued. The algorithm is entirely based on the availability of operands and functional units. However, older instructions have higher priority over younger instructions.

As the instructions carry the physical register identification, encoded in the rename phase, it is not necessary to access the mapping tables to verify that the source registers are ready or not. As described in the rename (section 5.4), each entry in the register file has a ’ready’ bit assigned to it. The ready bit is turned on when an instruction writes into the register and remains on until the register is released. Any instructions reading that register will only access its value after the ’ready’ bit is turned on. If the ready bit is off, it means that the instruction that writes in that register (producer) has not yet been executed and thus the value stored cannot be used as source for another instruction (consumer).

5.7 Write Back

The write back mechanism updates the ’ready’ bit of a the destination register of an instructions that completes its execution. The functionality of this stage is identical in the base architecture and DCE.

As instructions are issued out-of-order and might have different latencies they might complete out-of-order. The write back stage will update as many ready bits as possible according to the bandwidth available in the order the instructions finish. Priority is also given to the oldest instructions when the bandwidth available is not enough to ready the destination registers of all instructions completing in a given cycle.

Nothing special is done for instructions that are not producer such as branches and stores. When an instructions that does not write into a register completes the write back simply marks the instruction as completed and ready to commit with no need to update the register file.

5.8 Commit

The commit is performed in order to guarantee that instructions will not be committed speculatively. Therefore the commit follows strictly the order in which the instructions were inserted in the ROB during the dispatch stage.

The commit will check the instruction at the top of the ROB, oldest instruction, and commit it when it has completed (completed bit turned on by the write back stage).

If the instruction is other than a branch, the commit will increment the commit pointer and release the previous destination register, if one was marked during the remapping of the instruction.

When the instruction committing has a previous destination register, it is guaranteed that no other younger instruction will read its previous destination register since all younger instructions should be reading the new destination register produced by the instruction that is committing. Furthermore, there are no older instructions as the instruction committing is the oldest in flight instruction.

If the committing instruction is a store, the store data will be written into the memory as stores are not executed speculatively. Loads though can be executed speculatively since they do not change the state of the architecture when and if they are issued from the wrong path.

The commit will then check the very next instruction and perform the same procedure as the ROB operates as a First In First Out queue (FIFO). If the instruction is a branch, the commit will proceed differently if the branch was predicted or predicated.

Invalid instructions, that were invalidated previously by a predicated branch that resolved either taken or not, as skipped and thus do not commit their results to the architectural state as their destination registers were already released and might even being used by newer instructions.

5.8.1 Committing a predicted branch

When the branch was predicted the commit needs to check whether the prediction was correct or not. If the prediction was correct, the commit increments the commit pointer and looks for the next instruction.

When the prediction was wrong, the commit will signal all other stages a flush. The flush will clean up all buffers and mapping tables and redirect the fetch to the correct target. The checkpoint state saved during the rename, overwrites the default mapping table and becomes the new and only active mapping table. This mapping table has exactly the same contents that were in the mapping table after the branch was remapped.

This works as a roll back to the context which was before the branch was renamed. Everything that happened after that is useless since it was done based on instructions that were fetched down a wrong path.

A special procedure assures that only the registers that are currently mapped in the recovery mapping table remain active. All other registers are released and the processing starts once again from the correct address.

All younger predicated branches are also flushed. No benefit is obtained from the predication of these branches since they were part of the wrong path.

This procedure though is identical to the reference machine except that in the reference machine all branches are predicted.

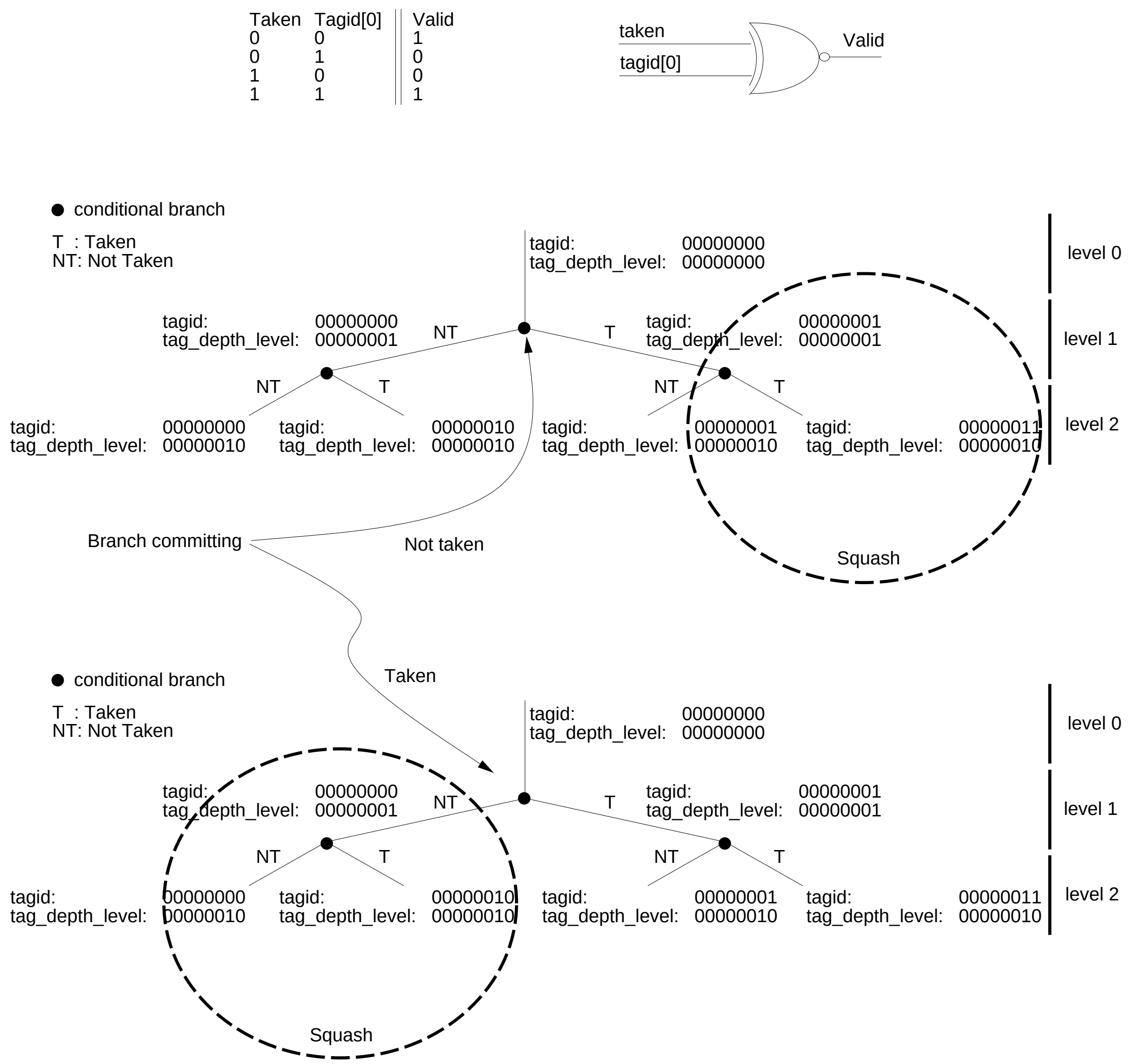
5.8.2 Committing a predicated branch

For predicated branches, the commit will execute a sequence of steps to squash the wrong paths that are in flight and were originated from the branch that is committing.

The first step is to invalidate the wrong paths. The commit broadcasts the information outcome information (taken or not) about the branch committing to other stages. All entries in the ROB and rename buffers contain a valid bit. The bit value is figured out through the logic presented in Figure 5.14.

Second, the tag_depth_level of all active tagids is shifted one bit to the right to adjust to the new level of each instruction. This process effects all entries in the ROB, rename buffer, TTB, mapping tables and register file. These two steps do not have to be serialized. As they are independent, they can be performed in parallel.

In some cases, specially when the distance between the branch and the exit point is



Taken	Tagid[0]	Valid
0	0	1
0	1	0
1	0	0
1	1	1

Figure 5.14: Tagid invalidation logic

large, it is possible that when the predicated branch commits, the exit point has not yet been renamed. In this cases, the branch is treated as if it was mispredicted since it would be very complicated to rename correctly the instructions that were already inside the pipe but not yet renamed if they don't have a tagid associated. A misprediction applies in this case to the branch committing even though it was predicated.

This decision was made in order to simplify the squash logic since it would require a more detailed monitoring of the instructions tagged. For example, if the branch resolving is not taken but the taken path is currently being renamed, although not entirely renamed, the rename would have to skip the instructions and go to the exit point. It is possible though the the fetch can be still fetching those instructions. So redirecting everything in order to skip only certain instructions would require a more complex logic to keep track of these situations.

5.9 Control Independence Data Independence Support

The rename in DCE operates in two modes similarly to the fetch. When the rename starts predicating a conditional branch, marked by the fetch, it switches to predication mode. When it sees the first instruction not marked for predication, that indicates that the instruction is the first one in the merge path.

The merge path is the first common instruction to all the paths started from that first conditional branch. The instructions in the merge path are executed regardless of the outcome of the branch and thus are called control independent.

Some of the instructions in the control independent path (CI) may use data produced in one of the paths of the branch being executed eagerly. In this case, DCE generates multiple replicas of the instruction, one per possible path so that all data dependencies are respected.

One of the problems in this approach is that several replicas of the same logical instruction are inserted in the pipeline generating some overhead. However, this overhead can be compensated by the fact that the instructions in the correct path are guaranteed to be useful when the branch commits. This avoids the misprediction penalties that would apply if the branch was mispredicted.

Although the extra instructions executed assure that the branch will not be penalized by mispredictions, the overhead introduced may slow down the correct path. If the overhead introduced is too large, the penalty introduced by slowing down the execution of the correct instructions may mask the potential benefit of avoiding the misprediction. If this happens, the benefit of applying predication may not compensate.

It is important to reduce the overhead to guarantee that the benefit of applying the predication will be larger than the penalty of possible mispredicting the branch.

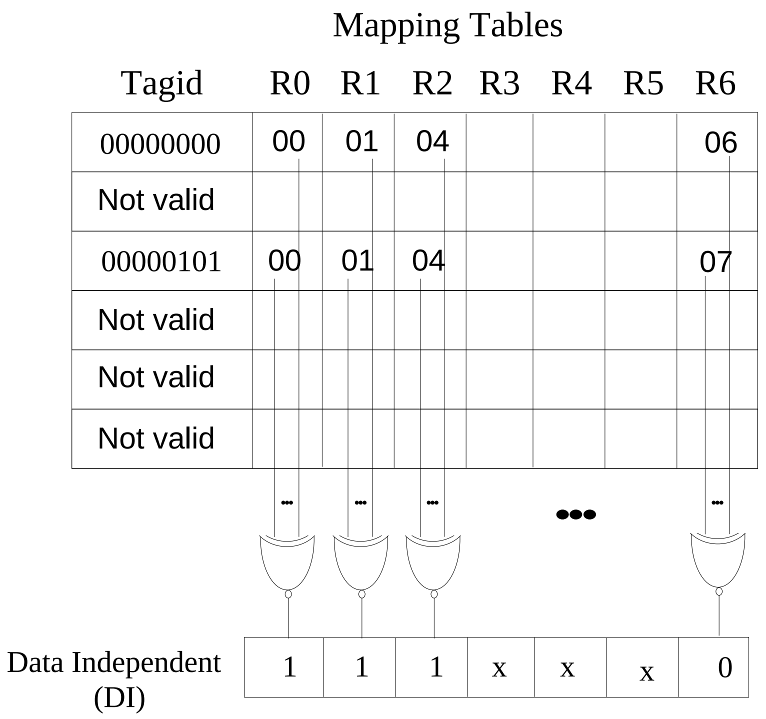


Figure 5.15: CIDI support

Certain instructions in the control independent path may not use data produced in one of the paths of the branch (data independent DI). The instructions may be laid in the control independent path even though they could have been executed actually before the branch. This happens because the compiler applies code motion techniques to reduce unnecessary register renaming, freeing up physical registers to be used by other instructions.

If the instruction is not used in one of the paths of the branch, the compiler may move the instruction delaying its execution if its produced result is not needed soon.

An instruction that resides in the control independent path and does not dependent on any data produced in one of the paths of the branch is called control independent data independent instruction (CIDI).

A CIDI instruction does not need multiple replicas as each replica would be exactly the same. In other words, the source physical registers mapped to the physical instructions would be the same.

If we apply the conventional renaming to these instructions, a physical destination registers will be allocated to each of them, even though the produced result will be identical to all of them. Furthermore there will be multiples identical replicas being executed.

To avoid this overhead, DCE can also detect this instructions by simply looking at the DI bit associated to each logical register in the mapping tables. Figure 5.15 presents the scheme used to detect DI instructions in DCE.

The tagging stage will generate multiple replicas of all instructions, as necessary, as it does not have access to the mapping tables. During the remapping, the rename will look at the DI bit associated to each of the source logical registers. If the DI bit is set for all of the sources, that means that the source logical registers are mapped to the same physical registers. In other words, the physical registers were not changed during the rename of any of the instructions in the branch's paths.

If the instruction has the predication bit off and the DI bit of all source registers is on, the instruction is called CIDI. The remapping detects that and even though there multiples replicas of the instruction, the remapping will send only one instructions down the path and it will cancel all other replicas created in the previous stage. The instruction will be marked as CIDI.

A CIDI marked instructions does not have a tagid associated to it. The instruction will not be squashed when a predicated branch resolves regardless of its outcome. As its in the control independent path, it is guaranteed that the instruction will commit except when a older predicted branch is mispredicted. In this case, all in-flight instructions are flushed. If all older branches are predicated, the CIDI instruction will always commit.

The destination register allocated for a CIDI instruction updates all valid mapping tables so its produced result can then be used by all consumer instructions the follow it.

5.9.1 Register releasing in the presence of CIDI

A slightly different approach is used to release the previous destination register of a CIDI instruction. Because a CIDI instruction may update several different mapping tables, DCE may need to released all previous destination registers of all mapping tables updated when the instruction was renamed.

A unique instruction identification number is assigned by the fetch unit to the instruction when it enters the pipeline. This unique number will be assigned to the physical registers removed from the mapping tables when the CIDI instruction updates them. In this case, the instruction does not carry the physical destination register number with it throughout its life, but instead, it leaves its identification number in the register file.

When the instruction commits, all physical registers marked with its unique identification number are released.

Although we described two different techniques to release the previous destination registers, we can potentially use the second technique to release any previous destination register.

6 EXPERIMENTAL ENVIRONMENT

In this chapter, we describe the experimental environment used in the simulations that are presented in the next chapter. It is important to highlight though that for the DCE results analysis, chapter 7, we used a more aggressive reference architecture than the one used in the profile analysis presented in chapter 3.

The reference architecture used from now on is based in the DCE pipeline. The main difference between the reference architecture and DCE is that DCE is capable of keeping multiple paths of predicated branches while the reference architecture only keeps one active path and predicts all branches. No predication is applied in the reference architecture.

6.1 Pipeline Structure

Both the reference architecture and DCE have 19 stages as shown on Figure 6.1. The stages named *Fetch, Rename 1, Rename 2, Dispatch, Issue, Execution, Write back* and *Commit* are modeled in the DCE simulator. The stages named *VIRTUAL x* are merely delays introduced to simulate a deeper pipeline and increase the misfetch and/or misprediction penalties in order to have a more realistic performance model similar to what is available in the state-of-the-art microprocessors nowadays.

The DCE simulator is based on Simplescalar 3.0 ((BURGER; AUSTIN, 1997)). There are several significant changes introduced in order to model the DCE architecture. Most of the changes were introduced to implement in a higher level of details the renaming, speculative loads, partial stores, a more realistic and conservative instruction cache access, selective squash, misprediction recovery and compiler support.

Each of the stages modeled was implemented strictly according the description presented in the previous chapter.

6.2 Benchmarks

In the simulations presented next, we used 7 of the SPECint95 benchmarks (Table 6.1). The only integer benchmark not considered was *Compress* as it did not presented any predicated branches in the simulations. The results of *Compress* are not presented and therefore its performance is not considered in the averages presented neither for the reference machine nor for DCE.

Float point benchmarks were not simulated as the number of mispredicted branches in most of them is not significant and does not overwhelmingly impacts the performance of those benchmarks. For some of the results, we presented only the benchmarks that presented highest misprediction rates as described in Chapter 3.

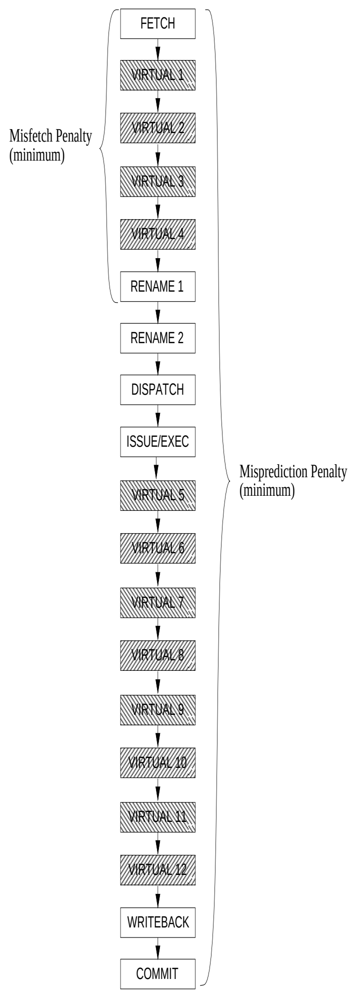


Figure 6.1: Pipeline Structure for Reference and DCE machines

We simulated several different configurations for each benchmark. For each configuration, we executed 1.3 billion instructions but considered only the last 300 million for the results. The first 1 billion instructions were executed in the fast forward mode and did not warm up the architecture. Only the main memory (not the caches) and the register file are updated with the results of these instructions so that a consistent context is provided for the execution of the remaining instructions. This artifact is used to skip a portion of the code.

6.3 Confi gurations

6.3.1 Reference Machine

Table 6.2 presents the configurations used in the reference machine. The configurations used allowed to simulate reference machines with Global or Local branch predictors and width of 8, 16, 32 or 64 instructions applied to the decode, dispatch, issue and commit.

8 simulations were performed for each benchmark varying the parameters mentioned above. As said previously, each simulation ran for 1.3 billion instructions totalizing 56 simulations (7 benchmarks * 2 predictors * 4 widths) and 72.8 billion instructions for the reference machine.

Table 6.1: Benchmarks Simulated

Benchmark	Input	Inst exec per confi g
Gcc	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -fi nline-functions -fschedule-insns2 -O cp-decl.i	1.3 billion
Go	50 21 9stone21.in	1.3 billion
Ijpeg	-image_fi le vigo.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.fi ndoptcomp	1.3 billion
li	*.lsp	1.3 billion
M88ksim	-c ¡ ctl.raw	1.3 billion
Perl	primes.pl ¡ primes.in	1.3 billion
Vortex	ref2.raw	1.3 billion

Notice that the functional units and memory ports were arbitrarily set to 32 each - a number that is not realistic. The intention behind this decision was not to create a primary bottleneck due to resource contention that could impact the analysis of the benefits of increasing parallelism by the employment of DCE. Furthermore, it is also important to emphasize that the reference machine is very aggressive in all of its configurations.

6.3.2 DCE Configurations (Set 1)

For DCE, we added 3 other parameters that are not used in the reference machine. First, the simulator has the ability to select what type of hammocks it can predicate.

The simulator allows to select simple (s), complex pure (c1), complex multiple joins (c2), complex multiple targets (c3) and/or complex overlapped hammocks (c4), which were the types presented in the compiler support section. The level of complexity associated with the predication of these hammocks increases according to the order in which they are shown here.

These options are cumulative. In other words, when complex pure is selected, that implies that simple is also selected. When complex multiple joins is selected, it implies that complex pure and simple are also selected and so on. Therefore, in the graphs presented in the next chapter, when simple is presented, that means that only simple hammocks were selected. When complex pure is presented, that represents simulations were both simple and complex pure hammocks were selected. The same applies for the other types of hammocks.

Second, the simulator permits to select the number of mapping tables to be used. Ultimately, the number of mapping tables specify the max number of outstanding paths active in the machine. We simulated configurations with 16, 32 and 64 mapping tables. TTBs were configure to support each number of mapping tables used.

Third, the distance from the branch to the target (not taken path) or from the target to the exit point (taken path) can be specified. We performed simulations with distances of 16, 32 or 64 instructions. The distance selected applies to each of the possible paths. Therefore, if a branch has 64 instructions in each of the paths, and the distance selected is 64, the branch will be allowed for predication even though it has in fact 128 instructions from the branch to the exit point. If the branch is one-sided, the distance applies to the not taken path which is then the distance between the branch and its target address.

Table 6.2: Reference Configuration

Resource	Confi guration
DCE	OFF
CIDI	OFF
Decode	8, 16, 32, 64 instructions decode bandwidth;
Rename	8, 16, 32, 64 instructions decode bandwidth;
Issue	8, 16, 32, 64 instructions issue bandwidth;
Commit	8, 16, 32, 64 instructions commit bandwidth;
Instruction window size	512 instructions
Load/store queue	128 instructions
Branch prediction	Global Hybrid: 2048 entries meta-table; 2-level gshare xor; 14 bits history;
	Local: 2-level local; 2048 history registers; 14 bits history;
Branch Target Buffer	512 sets; 4-way;
Return Address Stack	128 entries;
Integer ALU's (ialu)	32 Fus;
Integer mult/div (imult)	32 Fus;
FP ALU's (fpalu)	32 Fus;
FP mul/div (fpmult)	32 Fus;
Memory ports	32 ports;
L1 I-cache	2 cycle hit; 128 Kb (256 sets; 4 lines; 128 bytes line); LRU;
L1 D-cache	2 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU;
L2 Unifi ed cache	4 cycles hit; 2 Mb (2048 sets; 8 lines; 128 bytes line); LRU;
Main Memory	32 cycles fi rst chunk; 512 bytes bus;

Table 6.3 presents the configurations simulated in the first set of experiments. All 7 benchmarks were simulated with each of the configurations running for 1.3 billion instructions each.

In this set of experiments, we varied the complexity (s, c1, c2, c3,c4), number of mapping tables (16,32,64), distance (16,32,64), width (8, 16,32,64) and branch predictor (global, local). The icache line size was configured to support the distance selected for each simulation.

360 different configurations were simulated for each benchmark totalizing 2520 simulations and 3.276 trillion instructions retired.

6.3.3 DCE Configurations (Set 2)

In the second set of experiments, we fixed the distance to 64 and ran simulations with widths of 16, 32 and 64 and 128, 256 512 and 1024 mappings tables, varying also the complexity of hammocks selected for predication. We only simulated local predictor as this was the best branch predictor configuration obtained in the first set of experiments. Please refer to chapter 7 for more details.

We used the results of simulations from the first set for configurations of distance 64 and 16, 32 and 64 mapping tables.

Table 6.4 presents the configuration parameters used in the second set. This set of experiments was performed to expose the potential speed-up allowed by having a greater number of mappings tables and also to understand the overhead related to the increase of paths being executed.

60 simulations were executed for each benchmark totalizing 420 simulations and 546 billion instructions retired.

Table 6.3: DCE configurations (Set 1)

Resource	Confi guration
DCE	ON
CIDI	OFF
Hammock	simple, complex pure, multiple joins, multiple targets, overlapped
Mapping Tables	16, 32, 64
Distance	16, 32, 64
Decode	8, 16, 32, 64 instructions decode bandwidth;
Rename	8, 16, 32, 64 instructions decode bandwidth;
Issue	8, 16, 32, 64 instructions issue bandwidth;
Commit	8, 16, 32, 64 instructions commit bandwidth;
Instruction window size	512 instructions
Load/store queue	128 instructions
Branch prediction	Global Hybrid: 2048 entries meta-table; 2-level gshare xor; 14 bits history;
	Local: 2-level local; 2048 history registers; 14 bits history;
Branch Target Buffer	512 sets; 4-way;
Return Address Stack	128 entries;
Integer ALU's (ialu)	32 Fus;
Integer mult/div (imult)	32 Fus;
FP ALU's (fpalu)	32 Fus;
FP mul/div (fpmult)	32 Fus;
Memory ports	32 ports;
L1 I-cache	2 cycle hit; 128 Kb (256 sets; 4 lines; 128 bytes line); LRU;
	2 cycle hit; 256 Kb (256 sets; 4 lines; 256 bytes line); LRU;
	2 cycle hit; 512 Kb (256 sets; 4 lines; 512 bytes line); LRU;
L1 D-cache	2 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU;
L2 Unifi ed cache	4 cycles hit; 4 Mb (2048 sets; 8 lines; 256 bytes line); LRU;
	4 cycles hit; 8 Mb (2048 sets; 8 lines; 512 bytes line); LRU;
	4 cycles hit; 16 Mb (2048 sets; 8 lines; 1024 bytes line); LRU;
Main Memory	32 cycles fi rst chunk; 512 bytes bus;

6.3.4 DCE CIDI Configurations (Set 3)

Table 6.5 presents the configurations used in the third set of simulations.

In this set of simulations, the CIDI option was turned on. This option tells the simulator to use apply the CIDI mechanism when renaming instructions in the control independent paths of predicated branches.

CIDI instructions are renamed only once rather than multiple times when there multiple active tagids. The CIDI option reduces the overhead generated by the multiple paths by converging several replicas into one instruction that can be issued even before the originator predicated branch has resolved and that its result can be used by any instruction that follows even if the instructions are from different paths.

In this set, despite the use of CIDI, we simulated widths of 16, 32 and 64 with 16, 32, 64, 128, 256, 512 and 1024 mapping tables for distances of 64. We also varied the complexity of predicated branches (s, c1, c2, c3, c4).

For these simulations only *Go, Gcc* and *Ijpeg* were used as those were the SPECint95 benchmarks with highest misprediction rates since the benefits of DCE should be more visible where high misprediction rates occur as reducing this rates is the main objective of this technique. The reference averages used to calculate speed-up were also calculated

Table 6.4: DCE configurations (Set 2)

Resource	Confi guration
DCE	ON
CIDI	OFF
Hammock	simple, complex pure, multiple joins, multiple targets, overlapped
Mapping Tables	128, 256, 512, 1024
Distance	64
Decode	16, 32, 64 instructions decode bandwidth;
Rename	16, 32, 64 instructions decode bandwidth;
Issue	16, 32, 64 instructions issue bandwidth;
Commit	16, 32, 64 instructions commit bandwidth;
Instruction window size	512 instructions
Load/store queue	128 instructions
Branch prediction	Local: 2-level local; 2048 history registers; 14 bits history;
Branch Target Buffer	512 sets; 4-way;
Return Address Stack	128 entries;
Integer ALU's (ialu)	32 Fus;
Integer mult/div (imult)	32 Fus;
FP ALU's (fpalu)	32 Fus;
FP mul/div (fpmult)	32 Fus;
Memory ports	32 ports;
L1 I-cache	2 cycle hit; 512 Kb (256 sets; 4 lines; 512 bytes line); LRU;
L1 D-cache	2 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU;
L2 Unifi ed cache	4 cycles hit; 16 Mb (2048 sets; 8 lines; 1024 bytes line); LRU;
Main Memory	32 cycles fi rst chunk; 512 bytes bus;

using these 3 benchmarks only.

105 simulations were performed for each one of the 3 benchmarks totalizing 315 simulations and 409.5 billion instructions.

6.4 Validation

A validation process was put in place to validate the changes introduced in Simplescalar to model DCE.

The fastforward mode of Simplescalar executes instructions sequentially without simulating any parallelism. One instruction after another is simulated and there is no speculative instructions executed in this mode.

The original version of Simplescalar was modified to generate a dump of all instructions executed, the value of their source operands and the value produced by their execution. This dump produces the retired stream of instructions and therefore was assumed to be the golden model.

The DCE simulator should retire the same stream of instructions using the same set of operands source and producing the same results for each one. A perl script (dump_compare.pl) was written to launch the golden model and the DCE simulator and compare, instruction by instruction, all the source operands and the results produced.

The script detects any mismatches between the golden model and DCE, and starts a new run of DCE, with all the debug turned on, if a mismatch occurs. In the new run, DCE dumps information of each stage of the pipeline from a certain number of instructions

Table 6.5: DCE configurations (Set 3)

Resource	Confi guration
DCE	ON
CIDI	ON
Hammock	simple, complex pure, multiple joins, multiple targets, overlapped
Mapping Tables	16, 32, 64, 128, 256, 512, 1024
Distance	64
Decode	16, 32, 64 instructions decode bandwidth;
Rename	16, 32, 64 instructions decode bandwidth;
Issue	16, 32, 64 instructions issue bandwidth;
Commit	16, 32, 64 instructions commit bandwidth;
Instruction window size	512 instructions
Load/store queue	128 instructions
Branch prediction	Local: 2-level local; 2048 history registers; 14 bits history;
Branch Target Buffer	512 sets; 4-way;
Return Address Stack	128 entries;
Integer ALU's (ialu)	32 Fus;
Integer mult/div (imult)	32 Fus;
FP ALU's (fpalu)	32 Fus;
FP mul/div (fpmult)	32 Fus;
Memory ports	32 ports;
L1 I-cache	2 cycle hit; 512 Kb (256 sets; 4 lines; 512 bytes line); LRU;
L1 D-cache	2 cycle hit; 64Kb (256 sets; 4 lines; 64 bytes line); LRU;
L2 Unifi ed cache	4 cycles hit; 16 Mb (2048 sets; 8 lines; 1024 bytes line); LRU;
Main Memory	32 cycles fi rst chunk; 512 bytes bus;

before the mismatch until the instruction that caused the mismatch retires. A full run of both the golden model and DCE, without any mismatches, would prove that the simulator was performing the same architectural operations as the golden model.

The validation process was applied for each one of the benchmarks for at least one configuration. Figure 6.2 shows the validation model.

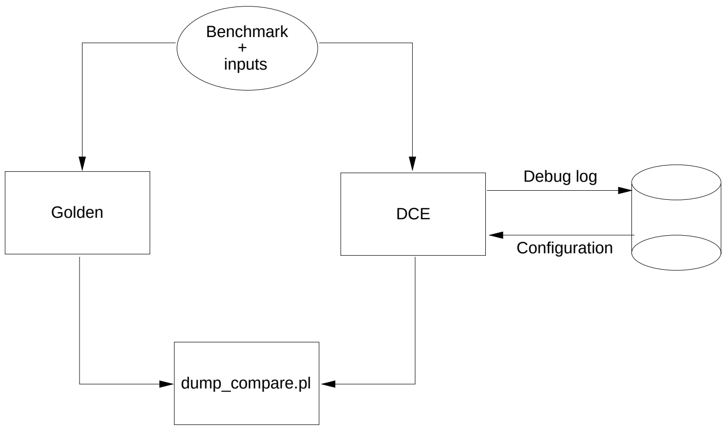


Figure 6.2: Validation model

7 RESULTS AND DISCUSSION

In this chapter we present the results of all simulations performed to evaluate the performance of DCE. In all graphs we compare certain aspects of DCE against a reference machine.

The reference machine consists of the same pipeline of DCE except that it predicts all branches and maintains only one active path per branch predicted. In DCE, some branches are predicated hence not requiring branch prediction. However, multiple paths of instructions are maintained.

7.1 Reference Machine's Performance

For the reference machine, we simulated 8 different configurations per benchmark. Figure 7.1 presents the IPC for each benchmark with each one of the configurations.

We varied the branch predictors used, Global or Local, and the machine widths in order to obtain a reference on the IPC indexes and how it improve when we change the predictor and/or increase the width.

The idea here is not to compare the performance of the benchmarks against themselves but yet create a reference and understanding of the impact of the variation of the different parameters mentioned above.

The main difference between the two predictors is that a Global predictor tends to perform better when branch correlation is crucial. The Local predictor keeps track of the history of individual branches providing less interference but it does not provide correlation history ((YEH; PATT, 1993)).

We can clearly see that the Global predictor works better for most of the benchmarks except *Go* where the Local predictor slightly outperforms the Global but the difference is not very significant. For *Perl* the performance is almost the same for any predictor.

The most interesting aspect is that the IPC does not increase significantly when we apply larger widths. Although we increase up to 64 the width of the most aggressive configuration of the reference machine the IPC does not increase proportionally to the potential added. This is mainly the result of the great limitation imposed by small basic blocks and the existence of branches in general and in special mispredicted branches as showed in chapter 3.

Vortex and *Ijpeg* presented significant increase in IPC when the width was increased from 8 to 16 instructions but it remained small when larger widths were applied. In overall, the average IPC was better using the Global predictor and in general larger widths did not provide much improvement as one would expect.

Another interesting result showed up in *M88ksim*. The best IPC was obtained with Global predictor and width of 8. This width outperformed all other widths with the Global

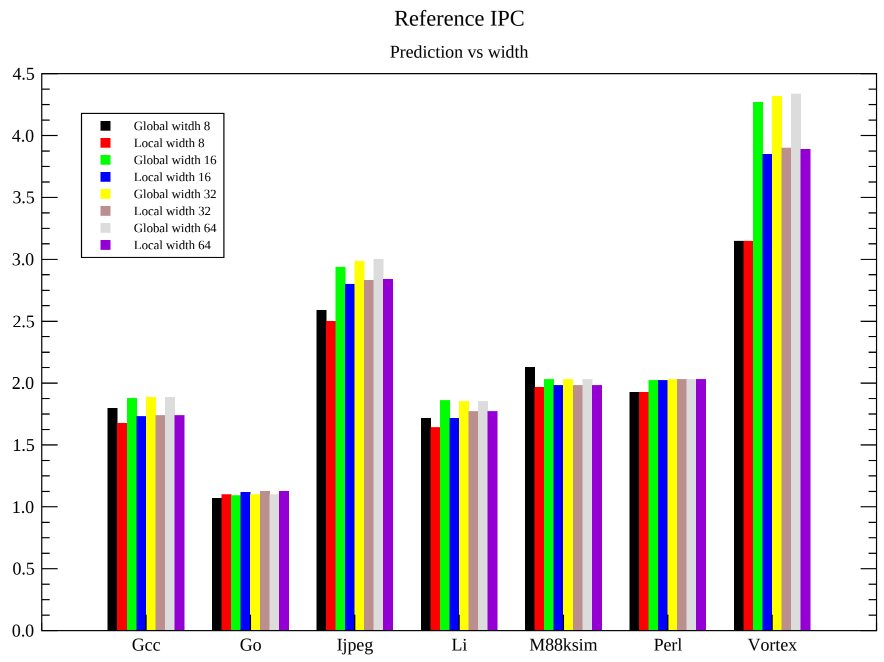


Figure 7.1: Reference IPC

or Local predictor. For the variations of the Local predictor though we still some small increase in the IPC when increasing the width but all of them are well below the width 8 with Global predictor.

A more detailed investigation showed that in this benchmark the number of mispredicted branches increased by around 20% with the Global predictor for widths larger than 8 instructions. This shows that some sort of interference occurred because more branches were predicted. For large widths, more instructions are fetched and executed per cycle. The number of predictions and updates to the predictor tends to be greater, and this seemed to have effected the performance of the predictor and consequently the overall performance of this benchmark.

The graph in figure 7.2 presents the harmonic mean of the IPC of all benchmarks simulated for each of the predictors and widths. The two right most bars present the Min and Max IPCs for all configurations. 1.81 instructions per cycle is the lowest IPC when the Local predictor was applied for width of 8, and 1.98 instructions per cycle was the highest IPC with Global predictor and widths of 32 or 64, an increase of about 9.3% with relation to the first.

Notice that the IPC does not increase linearly with the increase in width. There are several aspects that impose limitations to the IPC. These limits were demonstrated in chapter 3. Although the increase in width is fairly large from one configuration to another, the IPC increase is very small proving that the bottleneck is not in the bandwidth but instead in the available parallelism.

We will use the Global predictor configurations as reference from now on since the Local predictor did not outperform the Global predictor in any of the configurations. Even when we compare DCE with Local predictor, in the next sections, we will compare it against the reference machine with Global predictor.

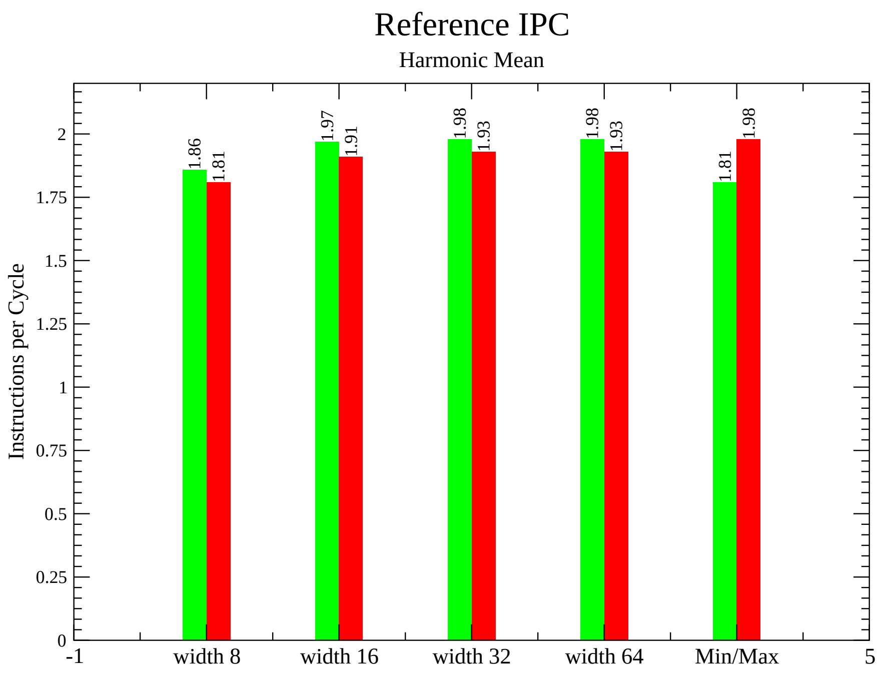


Figure 7.2: Reference IPC - Harmonic Mean

7.2 DCE IPC

In this section, we present the performance of DCE in the first set of experiments as described in details in the previous chapter.

The next three graphs present the DCE IPC for distances of up to 16 instructions in each path (Figure 7.3). The graph on the top presents the IPCs for the different predictors (Global or Local) and widths of 8, 16, 32 and 64 and 16 mapping tables. The graph in the middle presents the IPC for the same configurations but using 32 mappings tables. The graph in the bottom presents the IPC for configurations with 64 mapping tables.

The number of mapping tables increases from top to bottom and the widths increase from left to right. For each configuration we also present the IPC for the different types of branches selected for predication.

Notice that in most of the cases, the performance with the Local predictor is better than the performance with the Global predictor. This is just the opposite of what happened to the reference architecture.

In DCE, only a portion of the branches are predicted and thus only this portion updates the branch history register. Using a Global predictor is ideal when we want to benefit from branch correlation. We showed that that is the case for most of the benchmarks simulated when we presented the IPC for the reference machine.

Although the Global predictor performs better than the Local predictor when we predict all branches (in the reference machine), when we select branches that will not be predicted, we loose the history of those branches and therefore the correlated history becomes incomplete.

The incomplete branch history reduced the performance of the Global predictor in DCE. We see though that the Local predictor ends up performing a better job as it keeps track of individual branches and thus is not effected by the lack of history.

In the Global predictor, the history is updated speculatively. The prediction of a branch is based in the past execution of similar branches, with similar history, but it is also used

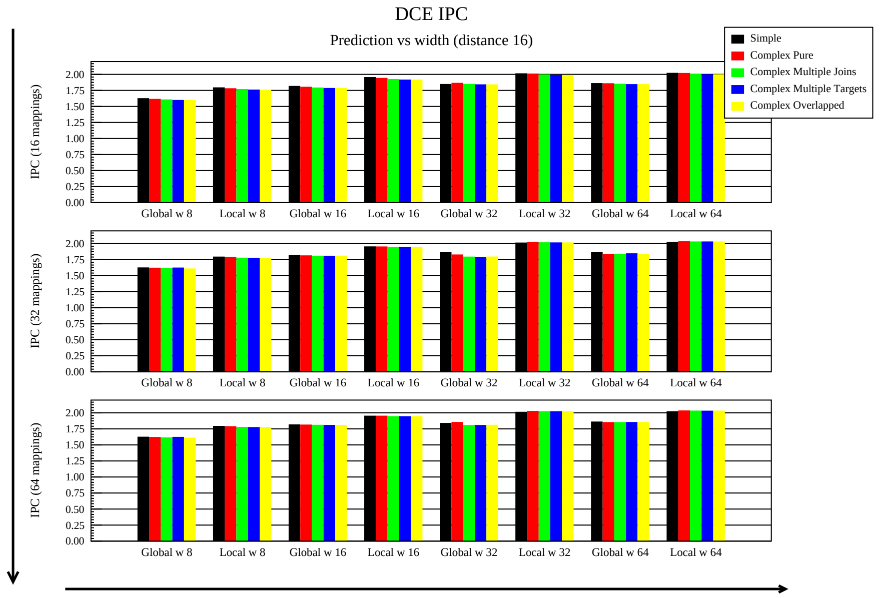


Figure 7.3: DCE IPC for distance 16

to updated the branch history register so that the next branch to be predicted can make use of the history of recently predicted branches.

In DCE, as we do not predict some branches, instead we execute both paths conditionally, we do not have a prediction and thus we cannot update the history register with accurate information.

In all simulations where we compare the performance of DCE against the reference machine, we will be using DCE with Local predictor and Reference with Global predictor. Because the overall goal is to compare the best performance of each approach, we decided to use the best predictor for each of the architectures so that we allow the best of each one to be compared against the other.

The IPC increases from left to right as we increase the width. Having a larger width allows a more aggressive execution providing more bandwidth. The IPC also increases from top to bottom as we add more mapping tables. More mapping tables will ultimately allow more paths to be kept active and thus more branches can benefit of the conditional execution.

When DCE runs out of mapping tables, branches that could be predicated are in fact predicted so that we do not have to stop the fetch unit until mapping tables become available. Because sometimes a branch will be predicted and sometimes will be predicated, according to the availability of mapping tables, when a branch that is usually predicated is instead predicted, the probability of a misprediction for that branch increases as possibly no history of that branch is available and if it is, it may not be up to date.

We also see that the IPC when we select only *Simple* branches is almost always better than the other cases. The exception is when the largest bandwidth is available with the largest number of mapping tables (64 mapping tables and width of 64). That the most aggressive configuration presented in this graph.

The reason is because all other branch's configurations add more complexity to the execution of multiple paths as there are nested branches whereas for *Simple* branches only, there are no nested branches in the predicated paths.

In any case, the configurations with width of 8 instructions did not outperform the reference machine. Although for configurations with widths larger than 16, the IPC starts

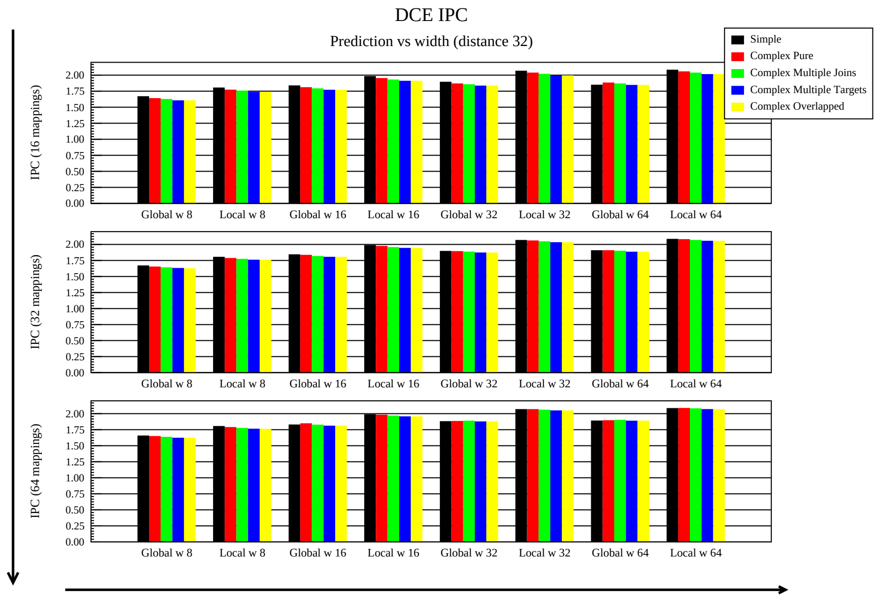


Figure 7.4: DCE IPC for distance 32

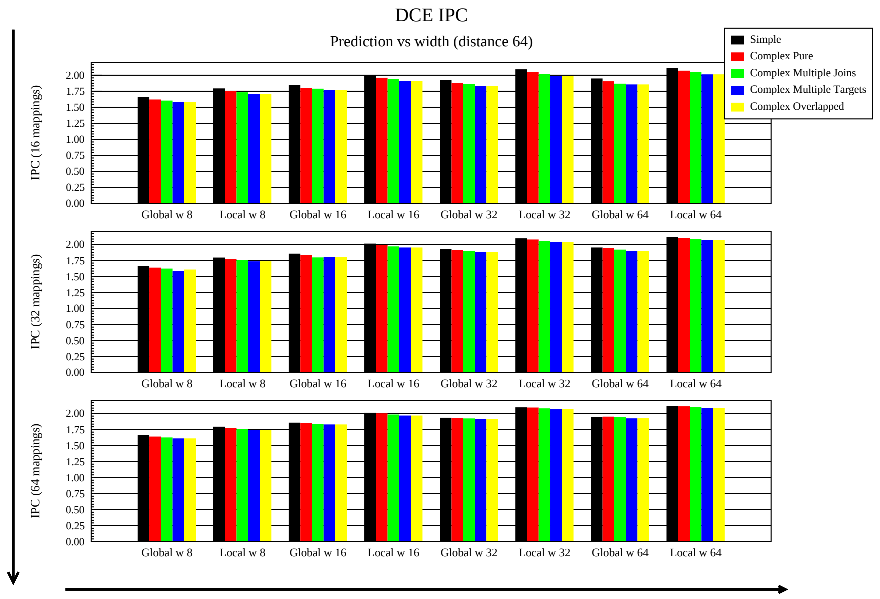


Figure 7.5: DCE IPC for distance 64

to achieve some gains over the reference architecture.

Increasing the number of mapping tables (middle and bottom graphs) allows the configurations of more complex branches to get close to *Simple*. This points out that to predicate more complex branches we need more mapping tables which is perhaps obvious since more complex branches will generate more outstanding paths hence more mapping tables are required.

When the configuration used has 64 mapping tables, the IPC for the different types of branches become flatter. In the results presented later, we increased the number of mapping tables beyond 64 to investigate the gains allowed by increasing this parameter.

Figure 7.4 presents the same configurations but for distance 32. Increasing the distance will allow more branches to be selected and thus the benefits of predication should also increase.

For distance 32, we see some increase in the IPC. However we still see that complex branches do not offer a better performance compared with *Simple*. We also start to see IPCs higher than 2 for widths of 16 and larger.

Figure 7.5 presents again the same configurations but applying distances of up to 64 for branch selection. It is interesting to see that in the first graphs (distance 16) we IPC was almost flat for the different types of branches and width 64.

In this graph, for distance 64 and 16 mapping tables, the IPC of the complex branches decreased significantly in comparison to the IPC of *Simple*. This happened because a larger distance allows more branches to be selected for predications. The number of complex branches tends also to increase thus requiring more mapping tables to actually realize some gain.

7.3 DCE Speed-up

Having discussed the graphs of IPC, we will compare the IPC of DCE against the IPC of the reference machine. As mentioned previously, we are comparing the IPC of the reference machine using a Global predictor and the IPC of DCE using a Local predictor.

The following graphs, as opposed to the IPC graphs, vary the distance and width for the same number of mapping tables. The first graph, Figure 7.6, presents the speed-up for 16 mapping tables. The speed-up is the percent increase, or decrease, in performance introduced by DCE.

The *y* axis represents the performance ratio of DCE in relation to the reference machine.

The predication of *Simple* branches only, provides a better ratio in all the cases with 16 mapping tables. We only see a real gain when the widths are larger than 16 for any type of branches in this configuration.

For *Complex* branches, we basically only see some gain when the width goes over 32, but they still do not outperform *Simple* branches. In average, for widths of 64, we see a performance increase of 2, 5 and 7%, respectively for distances of 16, 32 and 64.

Figure 7.7 presents the speed-up for 32 mapping tables. In this graph we see very similar ratios for *Simple* branches which confirms that increasing the number of mapping tables does not improve much the performance for this class of branches.

The best ratios are still achieved with *Simple* branches except for distances of 16 where now *Complex* branches outperformed *Simple* by a couple of points. We also see that the slow down introduced by predicating *Complex* branches is not as big as when we had only 16 mapping tables and, in general, it approaches very close the ratios of *Simple*.

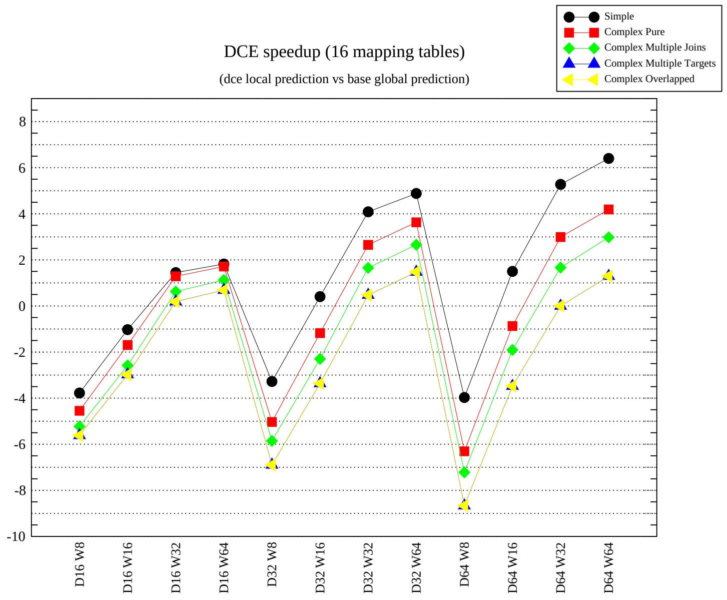


Figure 7.6: Average Speed-up varying predication distance with 16 mapping tables

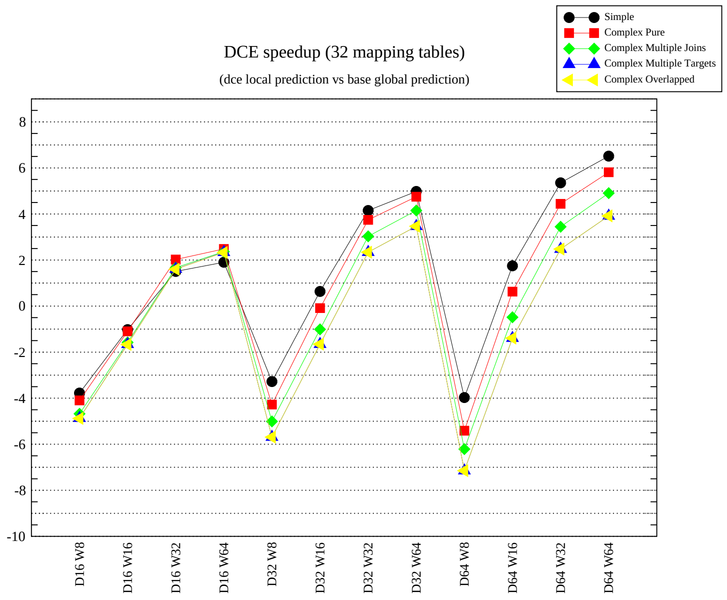


Figure 7.7: Average Speed-up varying predication distance with 32 mapping tables

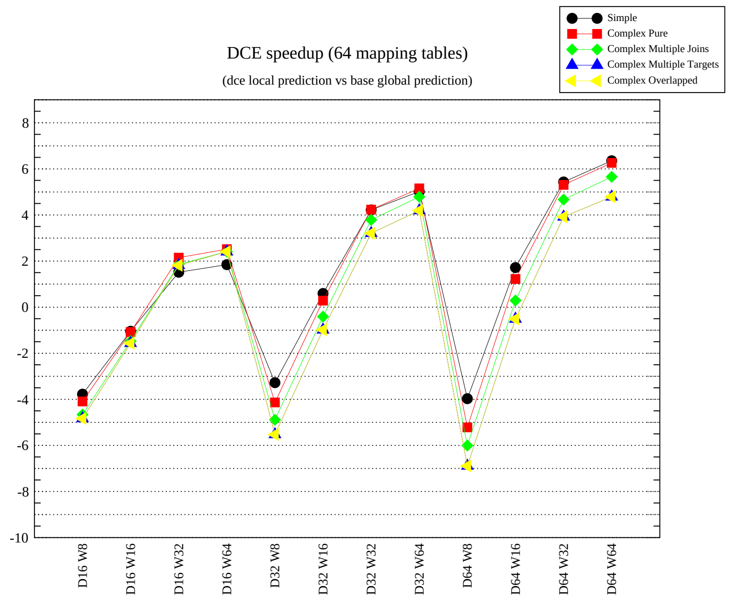


Figure 7.8: Average Speed-up varying predication distance with 64 mapping tables

When we increase the number of mapping tables to 64, Figure 7.8, the ratios for the different types of branches become very similar. In the IPC graphs we saw that the IPC was almost flat for configurations with 64 mapping tables.

An interesting point is that the overall performance dropped for distances of 64 when we increased the widths. This suggests that even though we probably can predicate more branches, the overhead generated by having more outstanding paths, degraded the potential performance.

7.4 Predicated Branches and Overhead Introduced

The following graph, Figure 7.9, presents the number of retired branches that were predicated in DCE. The direct benefit of predicating these branches is that they are virtually excluded from the total number of branches retired as there is no prediction and the fetch does not suffer disruptions regardless of their outcome.

The percentage of predications is given by the total number of retired conditional branches and the total number of conditional branches predicated. The graph in the top shows that in average from 5.8% to 7.3% of branches are predicated in DCE with 16 mapping tables.

As opposed to the IPC and Speedup graphs where *Simple* most of the time outperformed other *Complex* branches, here we can see that the total number of branches that are predicated can be increased by exploiting also complex branches.

With 16 mapping tables only, the number of predicated branches declines slightly when the distance is increased. The reason is because for larger distances the predication applied to a given branch will probably require the predication of a larger number of instructions in their both paths. The resources end up being allocated ultimately for a larger number of instructions but not necessarily for a larger number of branches as new branches that could be potentially predicated may turn out to be predicted.

The fetch can decide whether it will predict or predicated a branch based on the availability of resources. The preference is always given to predication if that is allowed for a

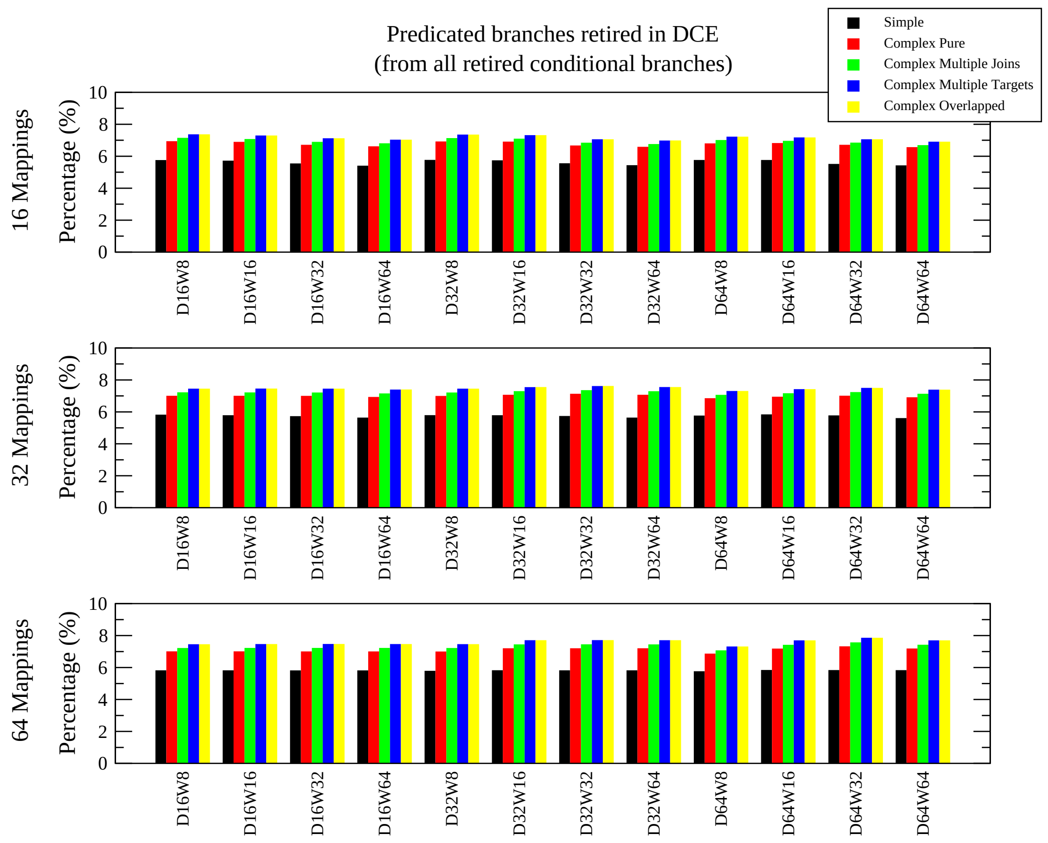


Figure 7.9: Percent of retired branches that were predicated

given branch and there resources available to starts the predication.

For 32 mapping tables, *Complex* branches also allow to predicate more branches than *Simple* although we saw that *Simple* still provides a better overall performance.

For 64 mapping tables, we actually see an increase in the number of predicated branches for larger distances that suggests perhaps increasing even more the number of mapping tables would allow more gains.

Although we see a higher number of predicated branches when we select complex branches, we do not see much improvement in performance. When we predicate a branch, we execute instructions from both paths in parallel. The resources are shared among these instructions.

In the conventional machine, these resource would be allocate exclusively for one of the paths. However, if the prediction is wrong, all operations that were performed speculatively are lost and bottom line is that the resources ended up being wasted.

In DCE this does not happen as it is guaranteed that one of the paths will be committed. There is a trade-off between chosen prediction or predication. If the prediction is wrong, the resources are wasted totally executing instructions from the wrong path. In predication, we assure one of the paths will be committed but it has to share the resources with the other path that will be squashed.

Figure 7.10 shows the overhead introduced by the predication. The overhead is calculated as the increase in the total number of instructions executed, but not necessarily retired, in relation to the reference machine. Many instructions are executed and flushed later because of mispredictions in both machines.

In DCE, in order to predicate branches, we end up executing more instructions than

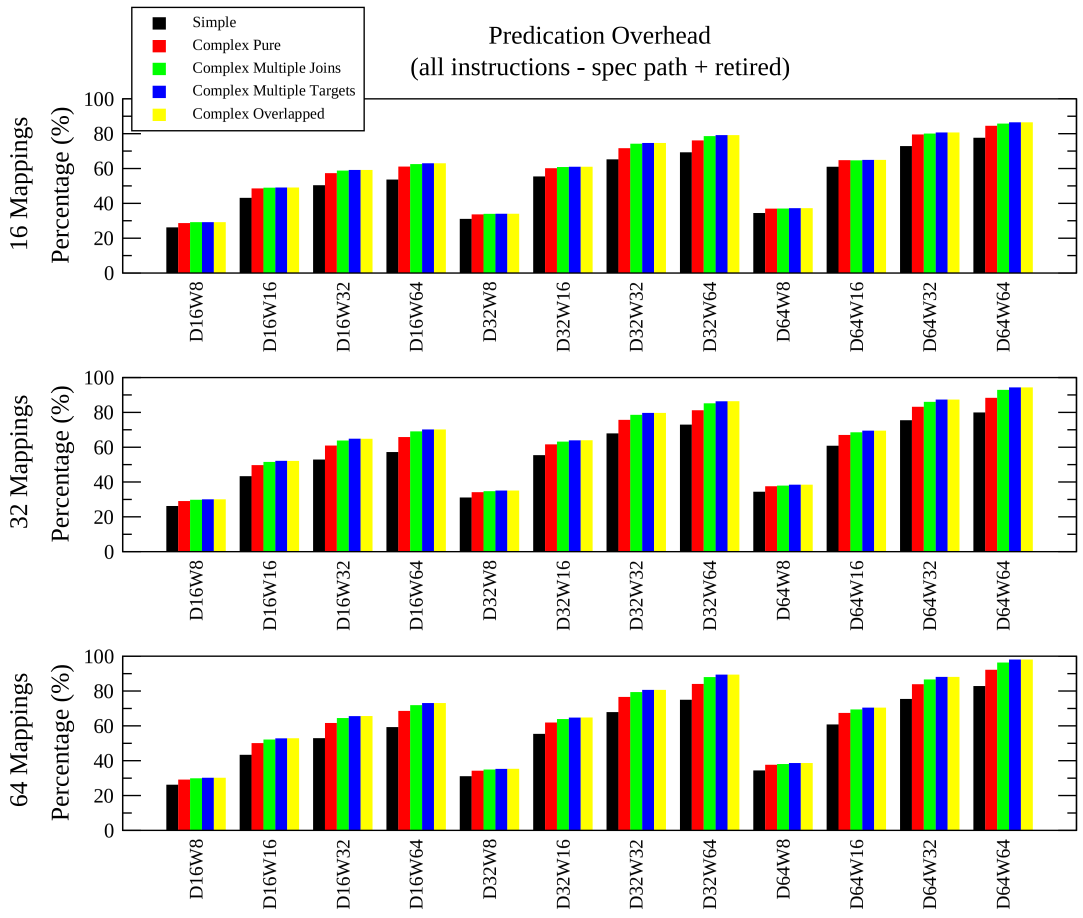


Figure 7.10: Percent increase in overhead due to predication

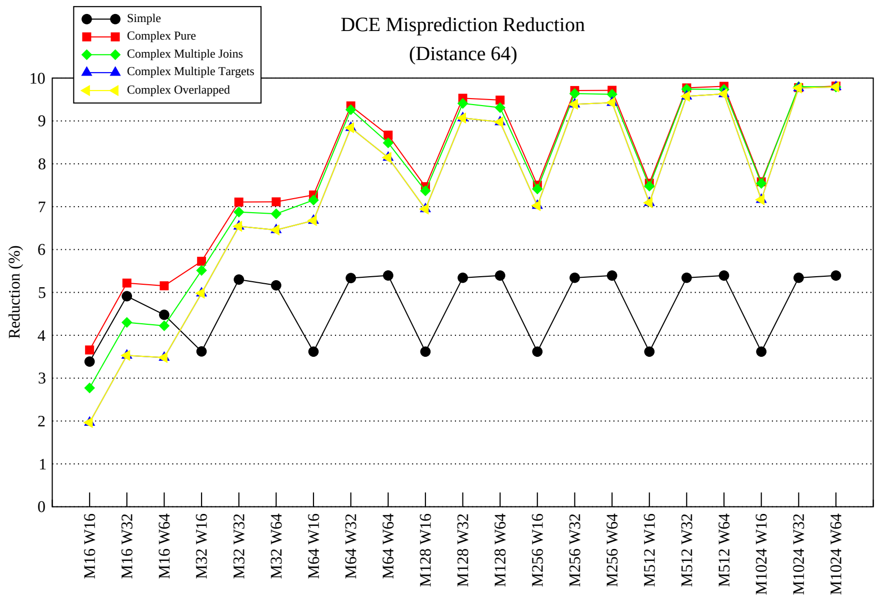


Figure 7.11: Misprediction reduction for distance 64

in the reference machine. The number of instructions executed in excess to the reference machine is very high. This is a huge performance limiting factor in DCE.

As expected, the overhead increases as we increase the number of mapping tables because it allows more branches to be predicated and also as we increase the width as it allows more instructions to be processed.

The worst case is when we configured DCE with 64 mapping tables, distance 64 and width 64. There was almost 100% more instructions being executed to provide an speed-up of around 5% to 7%. Reducing unnecessary overhead should provide more gains to DCE.

There is at least one way for reducing part of the overhead. The application of CIDI, described previously, can reduce the total number of instructions processed in the control independent paths. Later in this chapter, we will discuss in more details the application of CIDI in DCE.

7.5 Increasing Predication Depth

In this section we present the results for the second set of experiments. We fixed the distance for predication selection to 64 so that we can capture a larger number of branches for predication. Although increasing the distance adds complexity, the overall increase is not much compared with smaller distances.

In this set of experiments, we concentrated in the effect of increasing the number of mapping tables which leads to an increase in predication depth. The first graph, Figure 7.11, presents the decrease in mispredictions when we vary the number of mapping tables.

The number of mispredictions is measured at the commit of a branch. Branches that were predicated are always counted as a correct prediction and all other branches are counted according to what happened to their prediction.

A predicated branch can sometimes be treated as a mispredicted branch though. This was explained earlier and occurs when the predicated branch is ready to commit but the

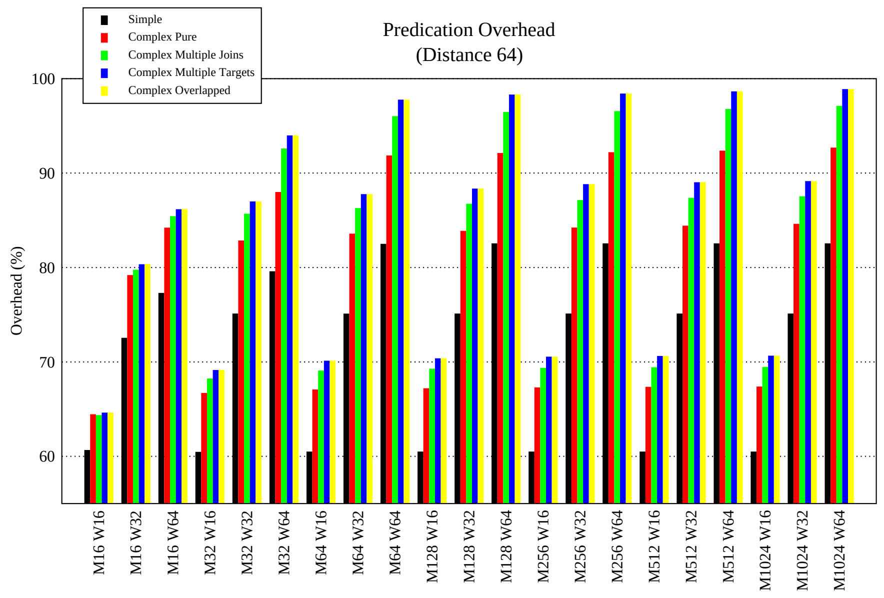


Figure 7.12: Predication overhead for distance 64

first instruction at the merge point was not tagged. In these cases, we revert the branch to a predicted type of branch and we take a misprediction penalty. Because of this, some predicated branch can still incur a misprediction penalty.

We vary the number of mapping tables from 16 to 1024. The width varies between 16, 32 and 64 and the distance is fixed at 64. We compare the different types of branches.

For *Simple* we see that the reduction in misprediction reaches, in the best cases, aroung 5.5% and has a very similar behavior for any number of mapping tables after 32, inclusive. This, one more time, confirms that to enjoy the max benefit of *Simple* branches we do not require a large number of mapping tables.

For all other complex branches, we see that the reduction is almost twice as much as for *Simple* branches reaching close to 10% for 1024 mapping tables. In most of the cases, we see that *Complex Pure* presents the best reduction rates. The misprediction reduction increases significantly from 16 trough 128 mapping tables and becomes almost flat after that.

Although we still see better rates after 128 mapping tables for complex branches, the benefit of adding more mapping tables is inhibited by the fact that the width limits the number of paths that can be created and thus a large number of mapping tables cannot be fully exploited with widths that are not proportional to the number of mapping tables.

In this graph is also demonstrated that complex branches can provide a lot more misprediction reduction than *Simple* branches. However this benefit is eclipsed by the overhead generated.

Figure 7.12 presents the overhead added by predication when mapping tables are increased.

The overhead exhibits a very similar curve as the misprediction reduction. It increases significantly up to 128 mapping tables and then it remains almost unchanged reaching almost 100% for configurations with *Complex Multiple Targets* and *Complex Overlapped*. And for *Complex Pure* it stays around 92% after 128 mapping tables.

We conclude here that the potential performance benefit of exploiting the predication is limited mainly by the bandwidth available and not by the parallelism created by having multiple paths active at the same time.

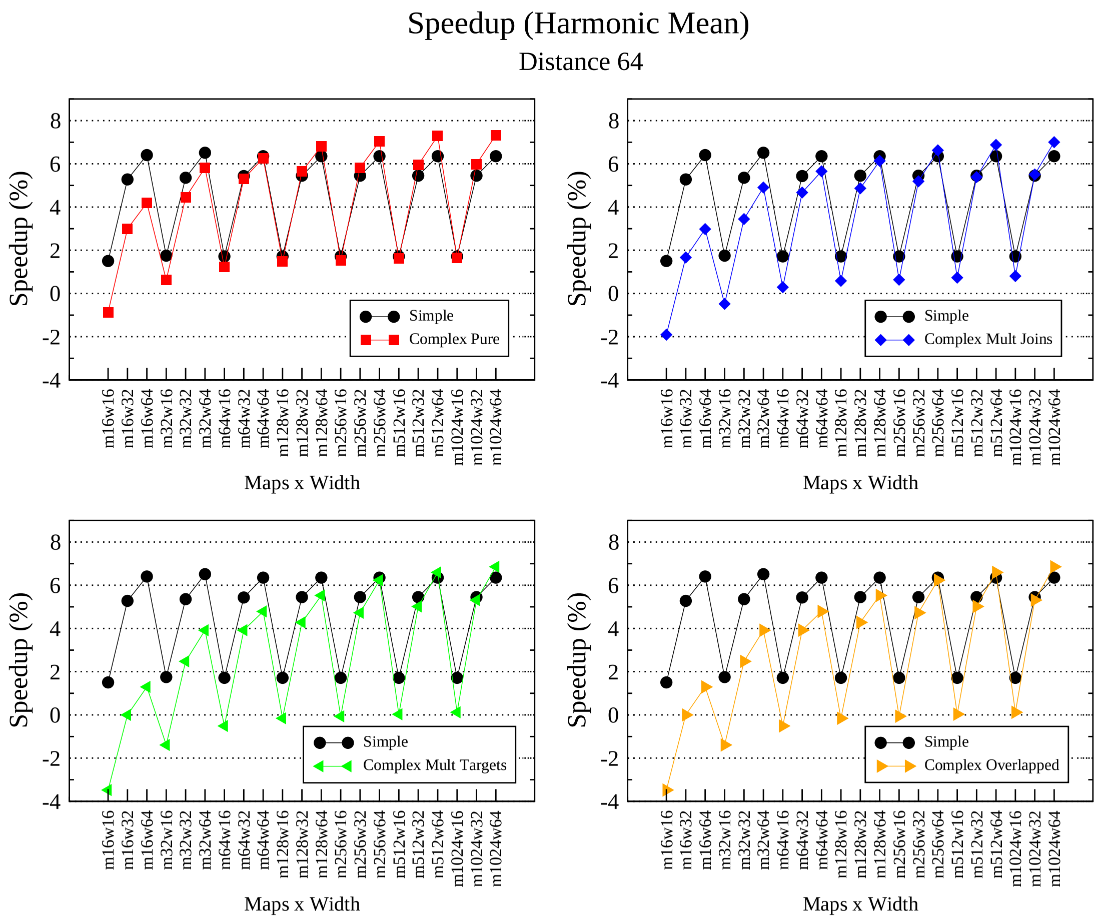


Figure 7.13: Average Speed-up for Distance 64 varying mappings tables

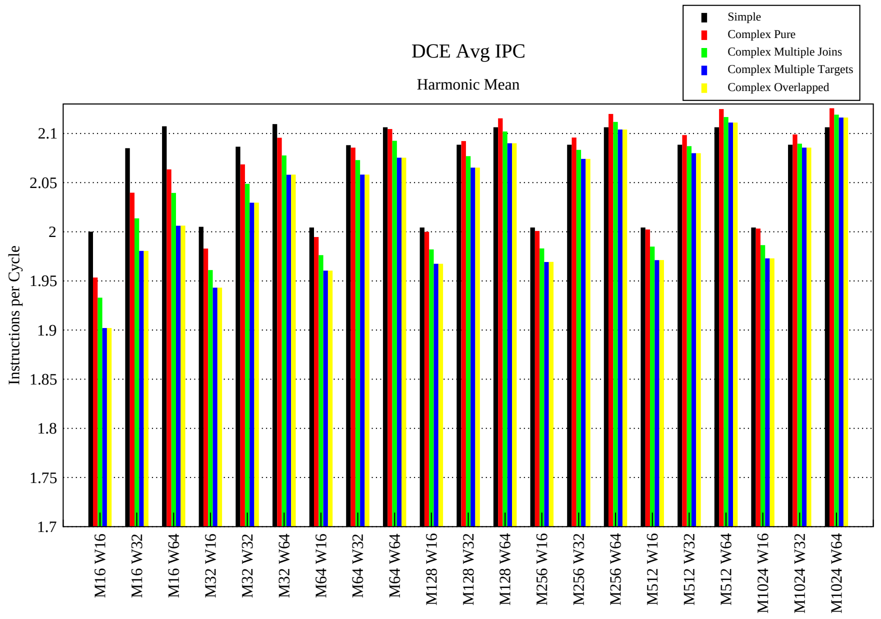


Figure 7.14: IPC Harmonic Mean - IPC varying number of mapping tables

The graphs in Figure 7.13 shows the speed-up for all mapping tables used in this set of experiments. We compare now the different types of branches against *Simple* branches and the speed-up of each type, including *Simple*, against the reference machine. The speed-up of *Simple* is the same in all graphs.

As said previously, *Simple* branches do not provide better performance when we increase the number of mapping tables. However, for complex branches, we can see that the number of mapping tables is crucial for a better performance.

The slow-down saw previously due to excessive overhead is seen here when the number of mapping tables is less than 64. From 64 mapping tables and up, the speed-up provided by complex branches is very similar to the one of *Simple* branches.

In the case of *Complex Pure* the speed-up is greater than *Simple* for configurations starting with 128 mapping tables but only for widths of 32 or more. But , in the best case, the difference is around 1% only.

Figure 7.14, shows the average IPC for all the configurations. We see clearly that *Simple* branches outperform complex in all configurations with less than 128 mapping tables, become similar when we have 256 mapping tables and complexes outperform *Simple* in configurations with 512 or 1024 mapping tables for widths of 32 or more.

At last, the conclusion from these experiments shows that predicating complex branches could provide a significant gain in performance given that the reduction in the misprediction rates is almost twice as much as for *Simple* branches. However the increase in overhead introduced by predicating more complex branches branches inhibit these gains.

7.6 Exploiting Control Independence Data Independence

In chapter 5 we described a mechanism for reducing the number of issued instructions down the control independent paths of predicated branches. CIDI, or Control Independence Data Independence was added to DCE as a form of compacting instructions that are in the control independent paths but do not dependent on conditional data produced in one of the paths of the predicated branches or one of its nested branches.

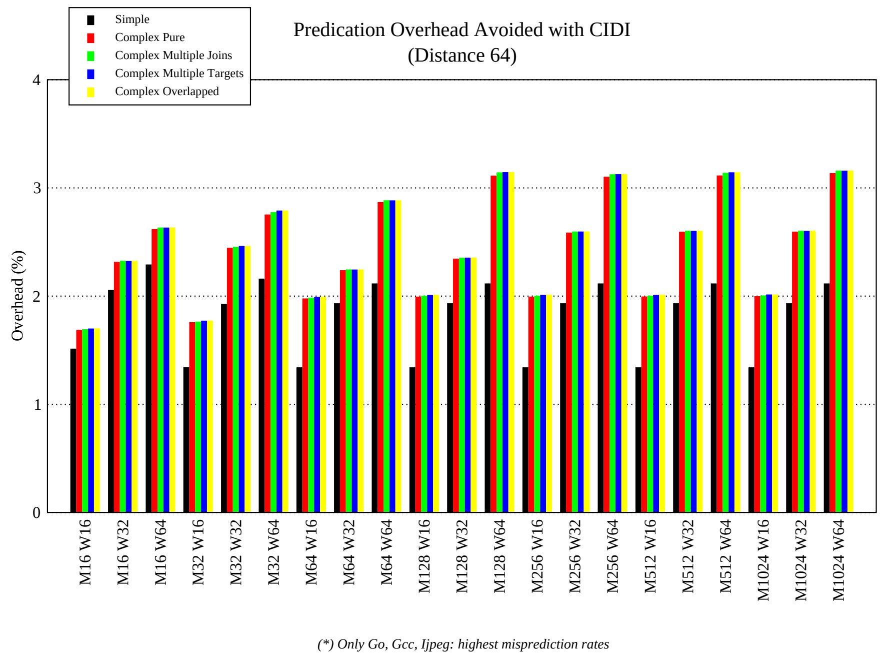


Figure 7.15: Predication overhead avoided when applying CIDI - distance 64

In this section we show the results of the third set of experiments. We simulated only *Go*, *Gcc* and *Ijpeg* in this set of simulations. The reason is that these are the three benchmarks that present the highest misprediction rates among all other benchmarks simulated in the first two sets of experiments.

DCE was designed to reduce misprediction penalties. The goal in this section is to observe the performance of DCE in these three benchmarks when we eliminate some of the overhead introduced by the predication.

By converging several replicas into one physical instruction, a portion of the overhead introduced by having multiple replicas of the same logical instruction can be eliminated. In DCE, a simple CIDI scheme was added to address this issue.

Although we can potentially apply CIDI techniques to converge any kinds of replicas into one instruction, in this set of experiments we show results of the application of this technique to instructions that are not control transfer, system calls or memory instructions.

Figure 7.15 presents the overhead avoided when we converged CIDI instructions. The overhead avoided is measured by counting the number of replicas of CIDI instructions eliminated.

When we are not using CIDI, the rename will created several replicas of any instructions in the control independent path of a predicated branch according to how many active paths exist in the moment when the instructions is renamed.

When using CIDI, if the rename detects that the instructions is in the control independent path and it is data independent, instead of sending *n* replicas of the instruction, one for each active path, it sends only one. We counted the number of replicas avoided and compared that with the total number of instructions executed in the reference machine. That is the number of replicas avoided when using CIDI.

In this graph, we can see that the overhead can be reduced by as much as 3.2% by converging regular CIDI instructions. The advantage of using CIDI could be potentially much higher if we would apply it also to memory instructions. In DCE, memory instructions also can create multiple replicas and thus cause contention in the memory system.

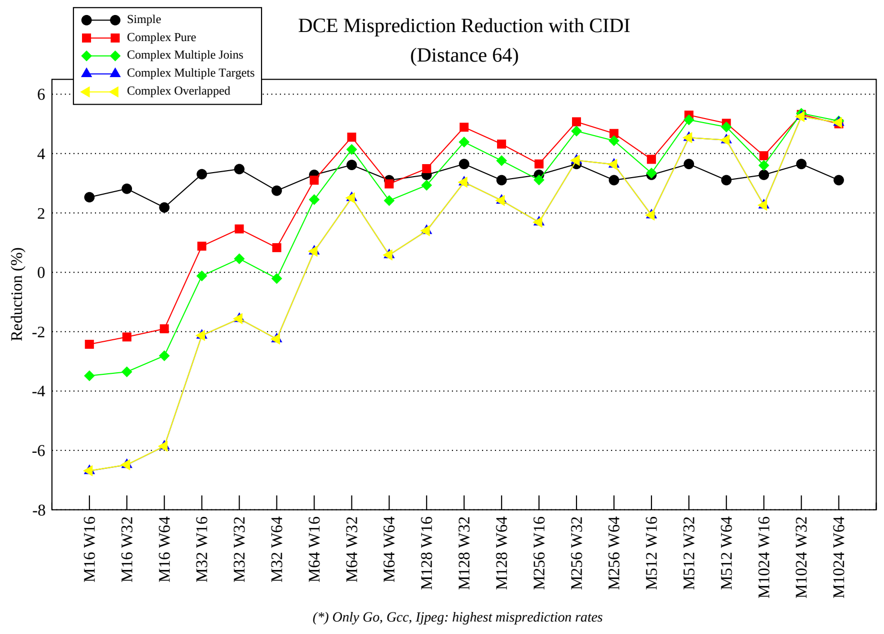


Figure 7.16: Misprediction reduction when applying CIDI - distance 64

If we could combine several replicas of memory instructions into one we would certainly reduce the bandwidth required to process these instructions as well as the latency required to bring the data for all of them. Although most likely only the first instruction that access the data may cause a cache miss, all subsequent accesses will take 2 cycles as that is the hit latency used in the simulations.

Also, because instructions may execute out-of-order, other instructions may remove the data from the memory causing replicas of a previous memory instruction that hit in the memory to cause a miss. We did not measured the interference caused by multiple conflicting memory replicas.

Graph 7.16 is presenting the misprediction reduction increasing the number of mapping tables for the three benchmarks selected for this set of experiments. Here it is possible to confirm that even small reductions in misprediction can provide nice increases in performance when misprediction is the limiting factor. Although the misprediction reductions is not as high as the average, the increase in speed-up provided is higher than the average speed-up because, in these three benchmarks, misprediction is the limiting factor.

The misprediction reduction is almost flat when we select only *Simple* branches as occurred in other simulations. The reduction for complex branches increases drastically from small number of mapping tables to larger numbers and outperforms *Simple* by as much as 55% for configurations with more than 128 mapping tables.

The speed-up provided by DCE in these benchmarks is well above the average. As said previously, these benchmarks are highly impacted by mispredictions therefore the benefits of DCE tend to be higher in their performance.

We also see here that complex branches require more mapping tables than *Simple* branches to show improvement. Although for *Simple* branches only the speed-up is almost constant throughout the simulations, for complex we see large increases when we apply more than 128 mapping tables. Actually even with 64 mapping tables we can see some significant improvement for *Complex Pure* branches.

In general, *Simple* branches achieved around 5% to 8.7% speed-up while *Complex Pure* reached almost 12%. This is almost 50% more speed-up than *Simple*. Between the

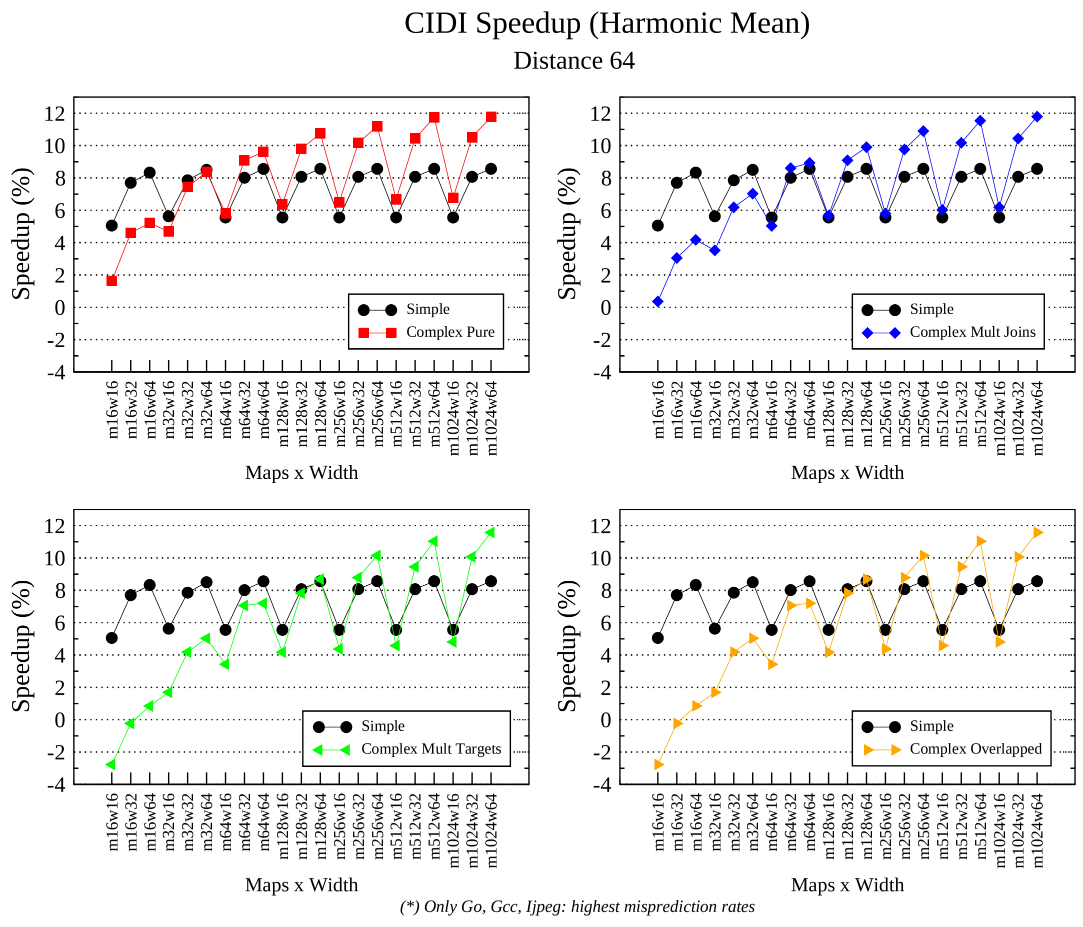


Figure 7.17: Speedup applying CIDI to reduce overhead - distance 64

different complex branches we did not see much difference proving that *Complex Pure* would be the best choice for these three benchmarks as it gives best performance with less complexity than the other kinds of complex.

7.7 Comparing DCE's Performance using Local Predictors

In all the previous graphs presented in this chapter we compared the performance of DCE using a Local branch predictor to predict the outcome of branches that were not selected for predication while in the reference machine a Global predictor was used.

It is well known that a Global in average performs better than a Local predictor (YEH; PATT, 1991, 1992, 1993). The performance improvements of a Global predictor, with relation to a Local predictor, are obtained from the ability of the Global predictor to capture the history of correlated branches whereas a Local predictor only benefits from individual history.

The initial experiments with DCE showed that the Global predictor used did not perform well. The reason was that the history of predicated branches was lost as there was no clue as to the outcome of these branches at fetch time. The Global predictor was updated with speculative history and so only the history of predicted branches could be inserted in the global history register.

It was proved later that even with a Local predictor, DCE offered performance speedups of up to 7.8% compared with the performance of the reference machine using a Global predictor.

While it is fair to say that the Local predictor was the best predictor for DCE, the Local predictor still it is not better than a Global predictor since it does not capture correlation information about the branches being executed.

We cannot say though that a Global predictor could not perform well in DCE. The Global predictor as used did not perform well in DCE as it could not benefit from a complete correlation history as history of predicated branches was not available at the time the history register was updated.

Graph 7.18 presents the harmonic mean of speedups for all integer benchmarks using a Local predictor in both the reference machine and DCE. The speedup is increase by 3-3.5% when comparing the performance of both architectures using similar predictors.

The speedup goes up to 10% in the best cases and in the worst cases, when very complex branches are predicated, the speedup is slightly below zero (negative) but it is indeed better than the negative speedups of around 4% obtained when comparing DCE with the reference machine using different predictors.

Although we cannot confirm that similar improvements would be obtained if one was using a Global predictor in DCE without the negative interference caused by missing the history information of predicated branches, we can see however that the performance of DCE can also be improved by improving the branch predictor's performance.

One way to do that is to updated the history register with the history of committed branches only. In this case then predictable fetched branches would be predicted based on old branches only and would not benefit from recent (speculative) history. Talcott (TALCOTT; NEMIROVSKY; WOOD, 1995) though showed that the prediction accuracy of history based branch predictors remains fairly constant as older branch histories are used.

Graph 7.19 presents the speedups of DCE using Local predictor and CIDI compared with the reference machine using Local predictor only.

The speedup is again increased by around 3-3.5% and reaches up to 14.5% in the best

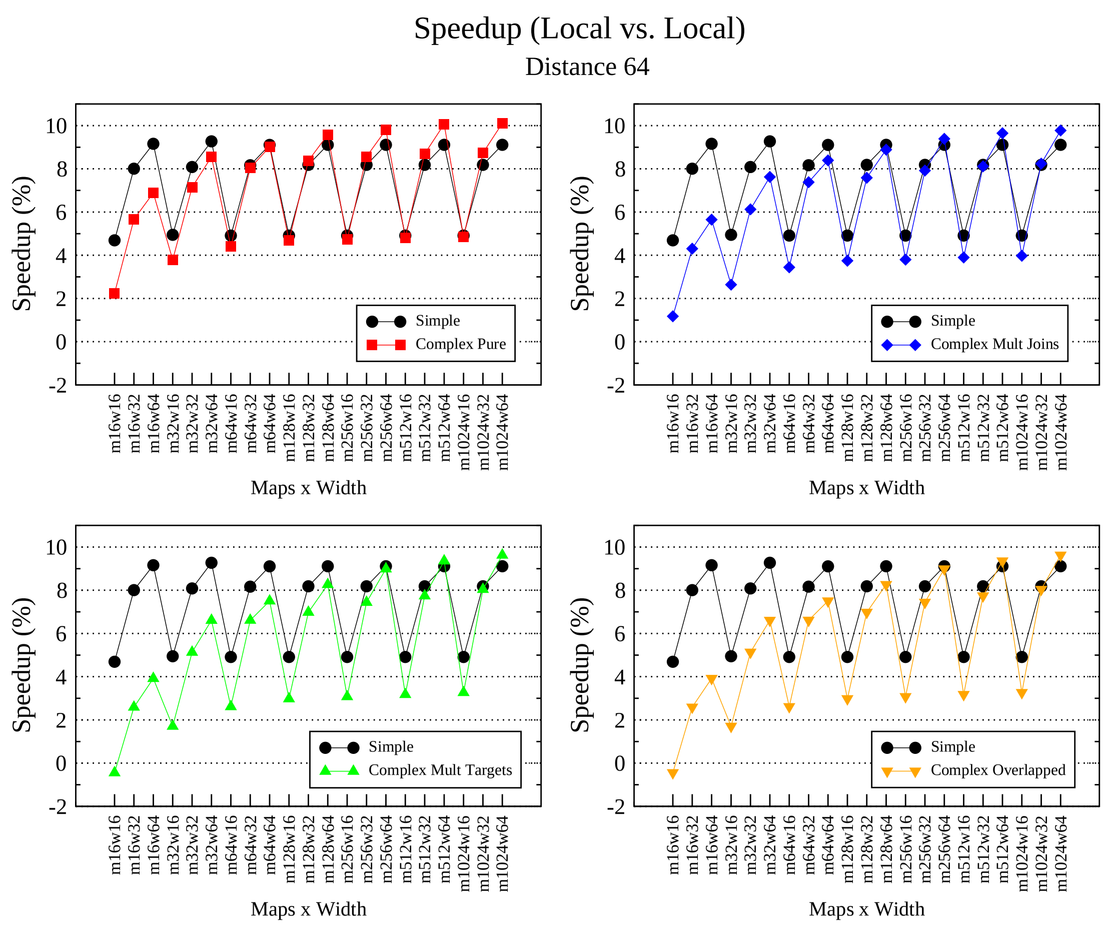
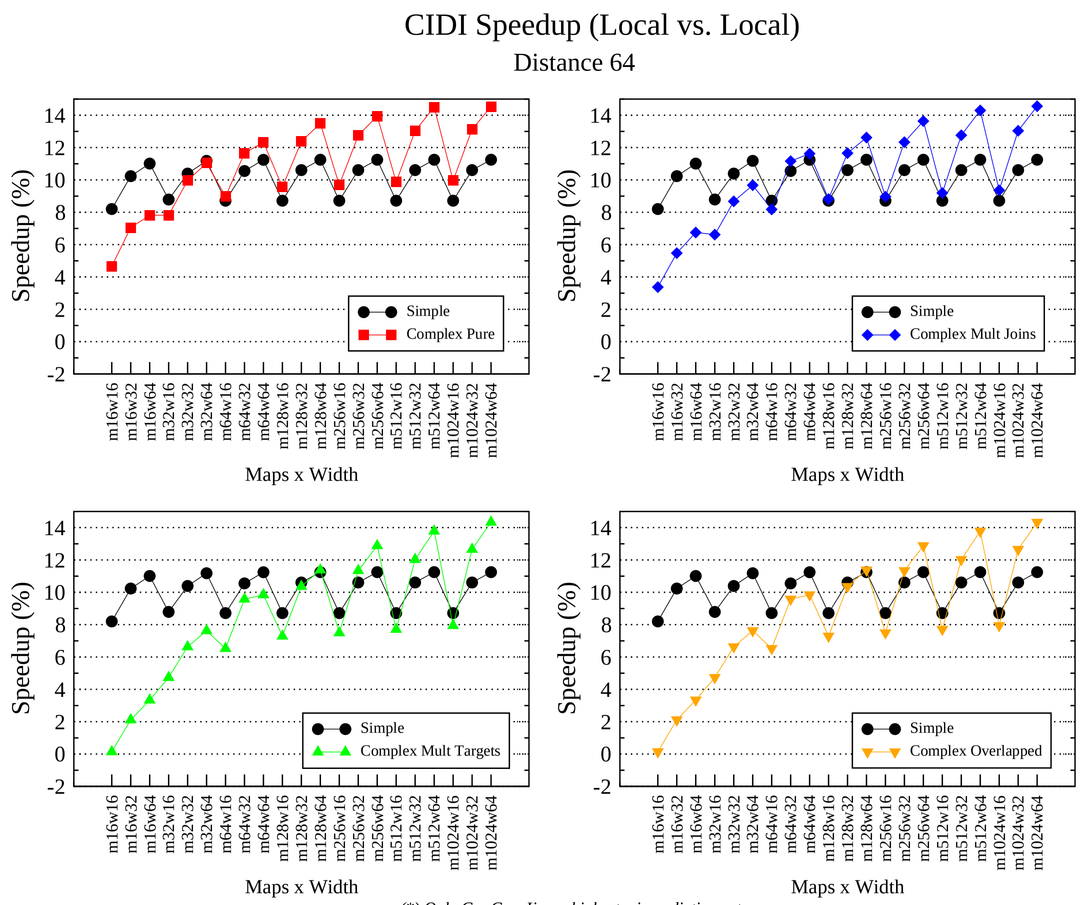


Figure 7.18: DCE speedup comparing Local vs. Local



(*) Only Go, Gcc, Ijpeg: highest misprediction rates

Figure 7.19: CIDI speedup comparing Local vs. Local

cases and is null in the worst cases when very complex branches are predicated and the overhead generated is very high.

8 CONCLUSION

In this thesis, we presented the Dynamic Conditional Execution scheme (DCE), a wide-issue superscalar processor that exploits the locality of conditional branches to apply dynamic predication and multipath. The result is a highly aggressive architecture that can reduce misprediction penalties and provide better overall performance than its superscalar counterparts that do not employ such mechanism.

The general concept behind DCE is to fetch both paths of some conditional branches sequentially based on a semi-static selection mechanism, execute both paths and dynamically and selectively squash one of the paths when the final outcome is available.

The major benefit of pursuing such approach is to reduce the number of predicted branches hence reducing the number of mispredictions without requiring special instruction set.

This work had a twofold goal. First, to analyze and understand the relationship between conditional branches, mispredictions and locality. Second, to propose and evaluate DCE, an architecture capable of exploiting the locality of some branches as a way to reduce misprediction penalties.

We showed initially that the performance of wide-issue superscalar machines is highly affected by branch mispredictions. The performance of these machines rely on the front end which has to be efficient in order to keep the level of parallelism required to fully exploit the wide machine parallelism.

We demonstrate that for a 12 stage 16 wide issue machine the IPC achieved is in average only 3.61 for all benchmarks, including float point. This is only 22.56% of what could potentially be achieve considering the amount of resources available.

Float point benchmarks are not as affected by mispredictions as integer benchmarks. For integer benchmarks, branch mispredictions are the most limiting factor to the fetch mechanism. Even when the branch is predicted correctly but if it is predicted taken, the fetch disruption also limits the number of instructions fetched per cycle.

In 66% of all benchmaks, the misprediction is the major limiting factor for the front end even when the predictor used provides high accuracy. When mispredictions happen, the number of executed instructions is very high compared with the number of instructions that commit and thus the efficiency of the machine declines. In integer benchmarks only 56.89% of instructions executed end up committed.

In order to determine the relationship between branches, mispredictions and locality, we also presented a branch profile study where provide a distribution of distances between the branches and their taken targets and the distribution of mispredictions based on the distances.

To conduct the experiments we configured a conventional superscalar machine with a gshare predictor with 12 bits of history and run simulations for integer benchmarks. We then analyzed the number of direct conditional branches that fall within distances of 4, 8, 16, 32 and 64 instructions from the branch to the taken target.

From this study, we concluded that 22% of branches have taken targets within 4 instructions from the branch and 36.57% of mispredictions are concentrated in these branches. Increasing the distance to 8, 34.43% of the branches will be within this distance with 48.42% of mispredictions occurring there. These numbers are even more appealing when the distances are further increased and for distance 16, 32 and 64, we see 57.14%,71.71% and 84.26% of branches with 65.86%, 83.57% and 93.00% of mispredictions, respectively within the distances presented.

Based on the branch profiling study, we proposed DCE. DCE is a wide-issue superscalar machine that can execute eagerly certain branches that fall within certain distances. DCE is a mix of dynamic prediction and multipath that reduces the complexity and disruptions of the fetch by fetching sequentially through branches that qualify for predication.

To determine if a branch qualify for predication, we also developed an extension of the selection mechanism proposed by (KLAUSER et al., 1998). In their selection mechanism, only simple branches would qualify for predication. We extended this model to also qualify complex branches and we further extended and divided the complex category into four different levels of complexity.

This selection mechanism is static and runs at compilation time marking branches that can be predicated. The fetch engine will decide whether a selected branch can be predicated or not based on the availability of resources.

The main difference between Klauser et al. (KLAUSER et al., 1998) and DCE is that DCE predicates both simple and complex branches and it does not use conditional moves to satisfy data dependences at the join point of predicated branches.

In Klauser et al., conditional moves are inserted dynamically at the joint point to block the issue of instructions from the same data chain of the predicated paths. When the branch resolves, the conditional moves can be issued to copy the data from the correct physical register to the correct source register. Thus, the original instruction that uses the respective register becomes ready for issue only after the conditional move instruction executes.

In DCE, a register renaming technique derived from Chaves et al. (CHAVES FILHO et al., 1999) generates replicas of instructions at the join point of predicated branches. Replicas are instructions that use data that is produced in one of the paths of a predicated branch. Therefore, there is one replica for each path predicated. As DCE does not use conditional moves, it does not block the issue of the dependent instructions. In DCE, the replicated instruction can be issued as soon as the the appropriate physical register is ready (i.e. the source data is available).

Therefore, DCE is a combination of a compiler selection mechanism and hardware to execute selected branches eagerly. In DCE, selected branches are treated as regular instructions by the fetch engine. They never disrupt the fetching as no prediction of control transfer is made at fetch time. In fact, no prediction is made for these branches at any time and that is why mispredictions cannot occur in these branches.

For a few reasons however, some predicated branches may be treated as mispredicted, but this occurs when a predicated branch once started, is found to not fit in the pipeline with the resources available at the moment. This characterizes an exception to the model and only occurs when the distances selected for predication are too large in comparison with the depth of the pipeline.

An execution-driven simulator was developed based on Simplescalar and a validation mechanism was put in place to validate the simulator's output. A golden trace is output from a simple functional simulator and compared with the trace generated by DCE. This was used to make the functional validation of DCE.

The implementation of DCE's architecture required some changes to the conventional machine. Most of the changes are concentrated in the renaming stages of the pipeline and are relatively simple. The rename basically supports multiple mapping tables. Each mapping table is assigned to a corresponding active outstanding path. Every time a new path is started, a new mapping table is generated with a copy of the previous active mapping table. Mapping tables are released when paths are squashed.

Although the implementation of DCE introduces some additional complexity to the rename stages of the pipeline in order to support multiple mapping tables, the fetch stage can potentially be simplified because less branches are predicted.

2741 simulations were performed with various different configurations totalizing around 3.5 trillion instructions simulated to evaluate the performance of DCE. The results showed that DCE indeed reduces the absolute number of mispredictions and therefore provides increased performance reducing also disruption in the fetch as less branches cause changes in the fetch flow.

The simulations in general showed that only 5-8% of retired branches were effectively predicated, even applying distances of 64. The initial profile studied suggested that a larger number of branches could have been predicated. Although the targets of these branches are indeed within relatively small distances, instructions in the body of these branches do not allow their predication in the architecture designed.

In the initial studies we only evaluated the distances. When implementing DCE, we also had to consider what instructions were contained in the branch's body. We cannot apply dynamic predication to branches which contain indirect, backward or far branches in their bodies. Then, even though a branch may have a target very near, one invalid instruction in its body may disqualify the branch for predication. This reduced considerably the number of branches predicated as the code was not optimized for this kind of predication.

In fact, this is a very important area for future research. We conduced all developments and experiments presented in this thesis based on conventional code generated by gcc. The code was optimized with standard optimizations techniques and was not changed to run on DCE. It is certain that optimization techniques targeted for DCE could increase the number of predicated branches thus increasing the potential benefits of DCE.

Our intent was to show that the changes required to implement DCE are relatively simple and that even conventional code can benefit from the dynamic predication technique proposed here without being modified.

Despite the few number of branches predicated, DCE offered an increase in performance of about 6 to 7.8% in average for integer benchmarks. The increase in complexity introduced by predicating complex branches did not pay off for small configurations and we concluded in these cases that predicating *Simple* branches would offer the best payoff.

The reason why complex branches do not offer better improvements is because the overhead generated by executing instructions from all possible paths of these branches is almost 100% more than in the conventional machine, for large configurations.

We showed though that despite the significant overhead introduced in DCE, we realized 6 to 7.8% speed-up in average while reducing the mispredictions by as much as 10%.

Based on the misprediction reduction we would expect a lot more improvement in terms of overall performance. However the overhead inhibits such benefits because resources end up allocated to instructions that are sometimes squashed. It is important to emphasize here that allocating resources for instructions that will be squashed slows down the correct path because it reduces the resources available, but it guarantees that the correct path always can be committed.

In conventional architectures, when a branch is mispredicted, resources are also allocated for instructions that will be squashed but this is totally wasted as it does not guarantee that the correct path can be commited. Furthermore, everything that is done, after a mispredicted branch is fetched, is flushed. Even the control independent instructions are flushed.

In DCE, the control independent path after a predicated branch is always committed as there is no flush associated with the execution of predicated branches. This certainly allows a smoother fetch flow.

The misprediction reduction for complex branches almost doubles when compared to *Simple* branches but the benefits of predicating complex branches are hidden because of the overhead.

Due to the overhead problem mentioned above, we also proposed a simple scheme to reduce overhead. This scheme was implemented in DCE and simulated showing a reduction of up to 3.2% of instructions in the control independent paths of branches.

Some instructions in the control independent paths are not dependent on data produced on previous paths of predicated branches. We combined this logical replicas into one physical instruction that can be utilized further in any of the paths that follow them. Minimal changes were introduced to DCE in order to pursue this approach.

A larger number of instructions could be eliminated in the CI paths if we apply this technique also to detect memory CIDI instructions. However in this case, more modifications would have been required and so this venue was not pursued further in this study.

The speed-up of complex branches showed to be almost 50% more than *Simple* branches when applying DCE with CIDI in the benchmarks (Go, Gcc and Ijpeg). These are the three integer benchmarks with highest misprediction rates in SPECint95 (excluding Compress).

The speed-up for this benchmarks was close to 12% with regard to the reference machine and the misprediction reduction was about 5.8%.

8.1 Future Research

There are at least two areas that deserve future and further investigation.

First, the compiler can be optimized to produce better code, more suitable for DCE. Optimizations should focus on creating shorter branches and removing invalid instructions from the body of those branches. This would increase the number of branches that can be selected for predication.

The compiler could also provide hints on whether selected branches should or not be predicated even when they qualify for predication under the restrictions defined by DCE in this thesis. In some cases, it might not be interesting to predicate some branches based on the target locality analysis only. The compiler can sometimes provide more information so that the processor can decide whether it starts a predication of a selected branch or not based on that information too and not only on the availability of resources.

Another optimization would be to move all DI instructions possible to the CI paths so that replicas of these instructions would be easily avoided by using the CIDI scheme proposed here.

Second, reducing the predication overhead can certainly lead to an overall better performance. One area that has shown promising results in a preliminary study, is instruction and trace reuse. Preliminary results have suggested that around 30% of instructions could be reused in DCE (SANTOS et al., 2003) and that reusable traces tend to be slightly larger in DCE than in conventional architectures.

Instruction reuse is in essence a non-speculative technique. Reusing instructions in the control independent paths of predicated branches can reduce significantly the overhead introduced and speed-up the processing of other instructions. Freeing up resources that could be allocated for instructions in the correct paths of predicated branches can certainly cause a major impact in DCE.

REFERENCES

AHUJA, P.; SKADRON, K.; MARTONOSI, M.; CLARK, D. Multi-Path Execution: Opportunities and Limits. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 12., 1998, Melbourne. **Proceedings...** New York: ACM, 1998.

ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992.

ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings...** New York: ACM, 1998.

ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 28., 1995, Ann Arbor. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995.

BURGER, D. C.; AUSTIN, T. M. **The SimpleScalar Tool Set, Version 2.0**. [S.l.: s.n.], 1997. (CS-TR-1997-1342).

BURGER, D. C.; AUSTIN, T. M.; BENNET, S. **Evaluating future microprocessors**: The Simplescalar toolset. [S.l.: s.n.], 1996. (CS-TR-1996-1308).

CHANG, P.-Y.; HAO, E.; PATT, Y. Alternative Implementations of Hybrid Branch Predictors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 28., 1995, Ann Arbor. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995.

CHAVES FILHO, E. M.; FERNANDES, E. S. The Effect of the Speculation Depth on the Performance of Superscalar Architectures. In: Euro-Par INTERNATIONAL EURO-PAR CONFERENCE, 3., 1997, Berlin. **Proceedings...** Berlin: Springer-Verlag, 1997. p.1061–1065. (Lecture Notes in Computer Science, v.1300).

CHAVES FILHO, E. M.; SANTOS, F. C. P.; SANTOS, A. D.; NAVAUX, P. O. A.; SANTOS, R. R. dos. MULFLUX: A Microarchitecture with Multiple Flows of Control. In: PROTEM-CC–PHASE I PROJECTS: INTERNATIONAL EVALUATION, 1999, Brasília. **Proceedings...** Brasília: CNPq, 1999. p.149–176.

CONTE, T.; N. MENEZES kishore; HILLS, P. M.; PATEL, B. A. Optimization of Instruction Fetch Mechanism for High Issue Rates. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 22., 1995, Santa Margherita Ligure. **Proceedings...** New York: ACM, 1995. p.333–344.

EVERS, M.; AL. et. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 23., 1996, Philadelphia. **Proceedings...** New York: ACM, 1996. p.3–11.

FRIENDLY, D. H.; PATEL, S.; PATT, Y. Alternative Fetch And Issue Policies for the Trace Cache Fetch Mechanism. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24., 1997. **Proceedings...** New York: ACM, 1997. p.24–33.

FRIENDLY, D. H.; PATEL, S.; PATT, Y. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 31., 1998. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p.173–181.

GWENNAP, L. Digital 21264 Sets New Standard. **Microprocessor Report**, [S.l.], Oct. 1996.

HEIL, T. H.; SMITH, J. E. **Selective Dual Path Execution**. Madison: University of Wisconsin, 1996. ECE Technical Report.

JACOBSEN, E.; ROTENBERG, E.; SMITH, J. Assigning Confidence to Conditional Branch Predictions. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 29., 1999, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.142–152.

JACOBSON, Q.; SMITH, J. Trace Preconstruction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 27., 2000, Vancouver. **Proceedings...** New York: ACM, 2000. p.37–46.

KESSLER, R. E. The Alpha 21264 Microprocessor. **IEEE Micro**, Los Alamitos, v.19, n.2, March/April 1999.

KLAUSER, A.; AUSTIN, T.; GRUNWALD, D.; CALDER, B. Dynamic Hammock Predication for Nonpredicated Instruction Set Architectures. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 7., 1998, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p.278–285.

KLAUSER, A.; PAITHANKAR, A.; GRUNWALD, D. Selective Eager Execution on the PolyPath Architecture. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings...** New York: ACM, 1998.

LAM, M. S.; WILSON, R. P. Limits of Control Flow on Parallelism. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992. p.46–57.

LEE, J. K.; SMITH, A. Branch Prediction Strategies and Branch Target Buffer Design. **IEEE Computer**, Los Alamitos, v.17, n.1, p.6–22, 1984.

MCFARLING, S. **Combining Branch Predictors**. Palo Alto: Western Labs, 1993. Technical Report. (DEC WRL TN–36).

MICHAUD, P.; SEZNEC, A.; JOURDAN, S.; SAINRAT, P. **Alternative Schemes for High-Bandwidth Instruction Fetch**. [S.l.]: IRISA, 1998.

MICHAUD, P.; SEZNEC, A.; JOURDAN, S.; SAINRAT, P. **Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors**. [S.l.]: IRISA, 1999.

PAN, S.-T.; SO, K.; RAHMEH, J. T. Improving the accuracy of dynamic branch prediction using branch correlation. In: SIGPLAN INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 5., 1992, Boston. **Proceedings...** New York: ACM, 1992. p.76–84.

PATEL, S.; EVERS, M.; PATT, Y. Improving trace cache effectiveness with branch promotion and trace packing. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings...** New York: ACM, 1998. p.262–271.

RAMIREZ, A.; LARRIBA-PEY, J.; VALERO, M. Trace Cache Redundancy: Red & Blue Traces. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 6., 2000, Toulouse. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000.

ROTENBERG, E.; BENNET, S.; SMITH, J. **Trace Cache**: a Low Latency Approach to High Bandwidth Instruction Fetching. Madison: University of Wisconsin-Madison/Computer Science Departament, 1996.

ROTENBERG, E.; BENNETT, S.; SMITH, J. A trace cache microarchitecture and evaluation. **IEEE Transactions on Computers**, New York, v.48, n.2, p.111–120, Feb. 1999.

SANTOS, R. R. dos. **Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções**. 1997. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SANTOS, R. R. dos; NAVAUX, P. O. A.; NEMIROVSKY, M. DCE: The Dynamic Conditional Execution Approach. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 7., 2001, Monterey. **Proceedings...** Los Alamitos: IEEE Computer Society, 2001. (Work in Progress Session).

SANTOS, R. R. dos; NEMIROVSKY, M. **DCE**: The Dynamic Conditional Execution approach in Multipath Control Independent Architecture. Santa Cruz, CA: Jack Baskin School of Engineering–UCSC, 2001. Technical Report. (UCSC–CRL–0108).

SANTOS, T. G. S. dos; PILLA, M. L.; SANTOS, R. R. dos; CHAVES FILHO, E. M.; BAMPI, S.; NAVAUX, P. O. A.; NEMIROVSKY, M. D. Resource Tuning in a Multipath Superscalar Architecture. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 11., 1999, Natal. **Proceedings...** Porto Alegre: SBC, 1999. p.27–34.

SANTOS, T. G. S. dos; SANTOS, R. R. dos; NAVAUX, P. O. A.; BAMPI, S.; NEMIROVSKY, M. Analyzing the Limits of Trace Reuse in a Dynamic Conditional Execution Architecture. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 17., 2003, San Francisco. **Proceedings...** New York: ACM, 2003. (Workshop: Exploring the trace space for dynamic optimization techniques).

SEZNEC, A.; JOURDAN, S.; SAINRAT, P.; MICHAUD, P. Multiple-block Ahead Branch Predictor. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEM (ASPLOS), 7., 1997, New York. **Proceedings...** [S.l.: s.n.], 1997.

SEZNEC, A.; MICHAUD, P. **De-Aliased Hybrid Branch Predictors**. [S.l.]: IRISA, 1999. (1229).

SKADRON, K. **Characterizing and Removing Branch Mispredictions**. 1999. PhD Dissertation — Princeton University, Princeton.

SMITH, J. E. A Study of Branch Prediction Strategies. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 8., 1981. **Proceedings...** New York: ACM, 1981. p.35–48.

TALCOTT, A.; NEMIROVSKY, M.; WOOD, R. The Influence of Branch Prediction Table Interference on Branch Prediction Scheme Performance. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 1995. **Proceedings...** [S.l.: s.n.], 1995.

UHT, A.; SINDAGI, V. Disjoint Eager Execution: An Optimal Form for Speculative Execution. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 28., 1995, Ann Arbor. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995. p.313–325.

UHT, A.; SINDAGI, V.; SOMANATHAN, S. Branch Effect Reduction Techniques. **IEEE Computer**, Los Alamitos, May 1997.

WALL, D. W. **Limits of Instruction-Level Parallelism**. Palo Alto: Western Research Laboratory, 1993.

YEH, T.-Y.; MARR, D.; PATT, Y. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 7., 1993. **Proceedings...** New York: ACM, 1993.

YEH, T.-Y.; PATT, Y. N. Two-Level Adaptive Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991, Alburquerque. **Proceedings...** Los Alamitos: IEEE Computer Society, 1991. p.51–61.

YEH, T.-Y.; PATT, Y. N. Alternative Implementations of two-level Adaptive Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992. p.124–134.

YEH, T.-Y.; PATT, Y. N. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 20., 1993, San Diego. **Proceedings...** New York: ACM, 1993. p.257–266.

APPENDIX A CONDITIONAL BRANCHES PROFILE

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

APPENDIX B SHORT BRANCHES DISTRIBUTION

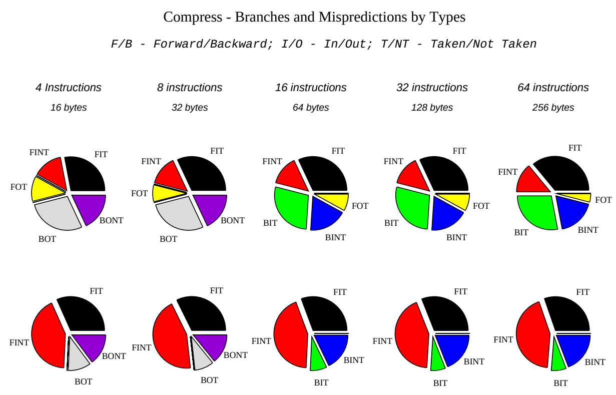


Figure 8.1: Short branches and Mispredictions - Compress

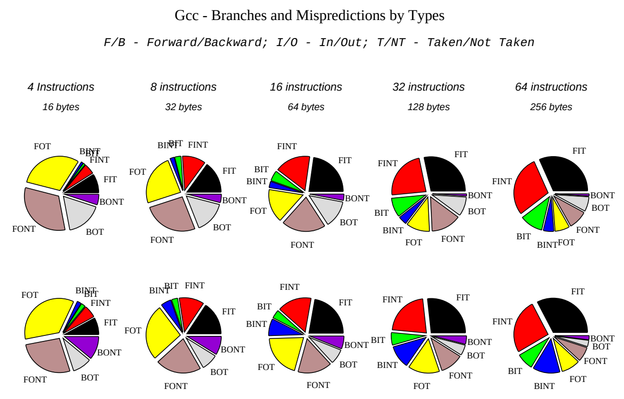
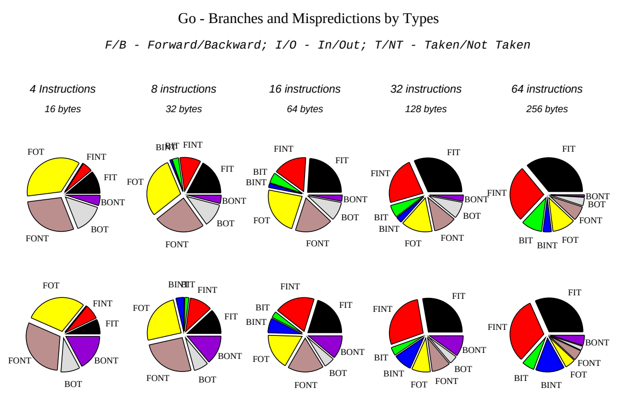
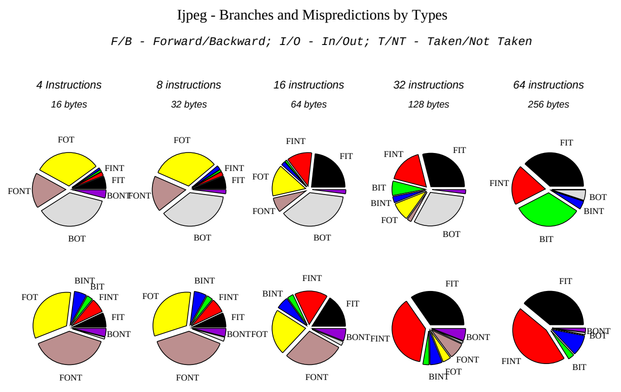
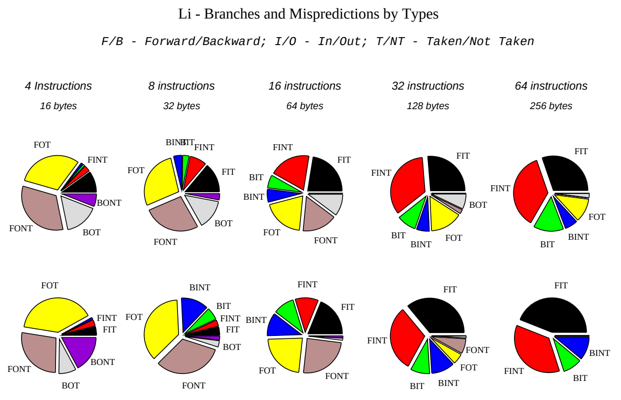


Figure 8.2: Short branches and Mispredictions - Gcc







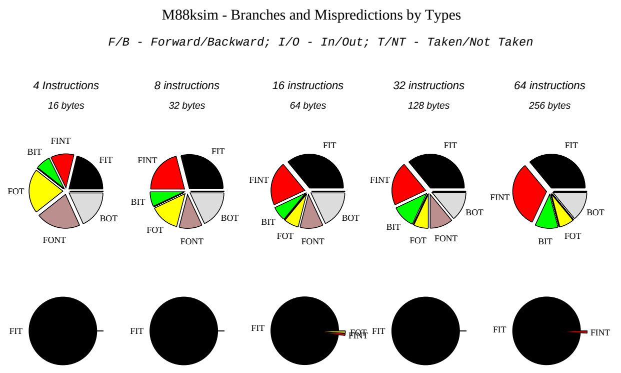


Figure 8.6: Short branches and Mispredictions - M88ksim

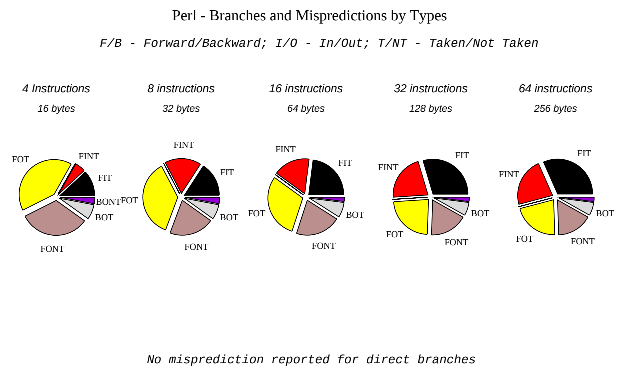


Figure 8.7: Short branches and Mispredictions - Perl

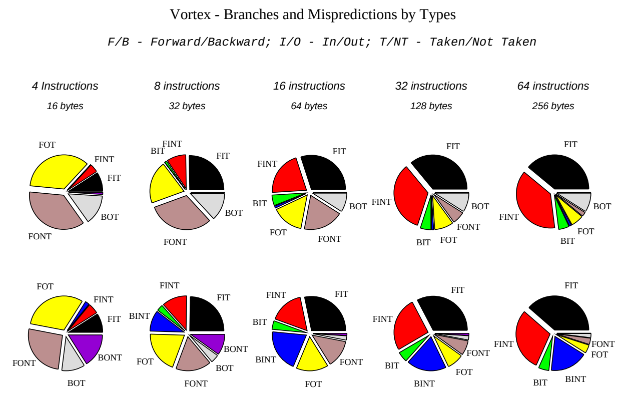


Figure 8.8: Short branches and Mispredictions - Vortex

APPENDIX C DCE IPC PER BENCHMARK

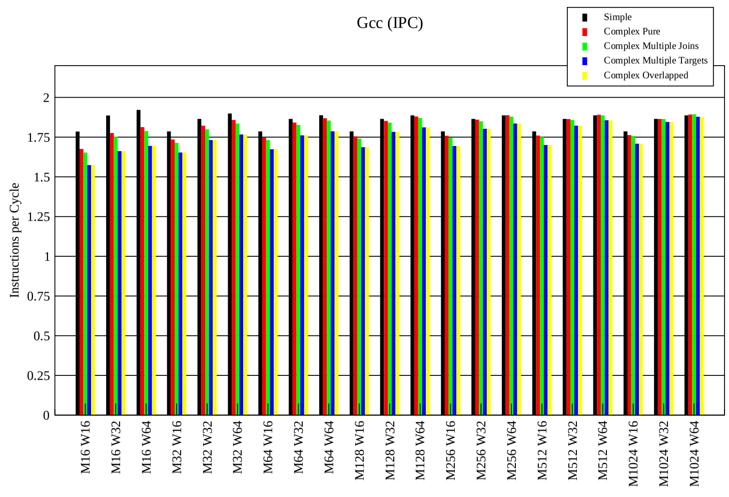


Figure 8.9: GCC - IPC varying number of mapping tables

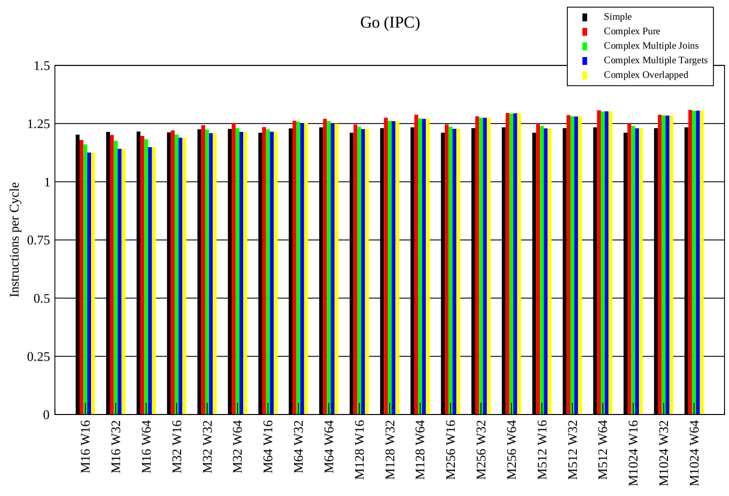


Figure 8.10: GO - IPC varying number of mapping tables

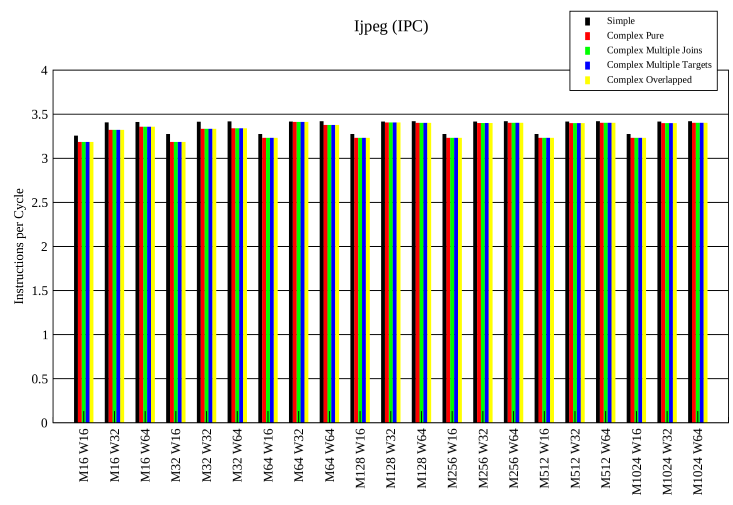


Figure 8.11: Ijpeg - IPC varying number of mapping tables

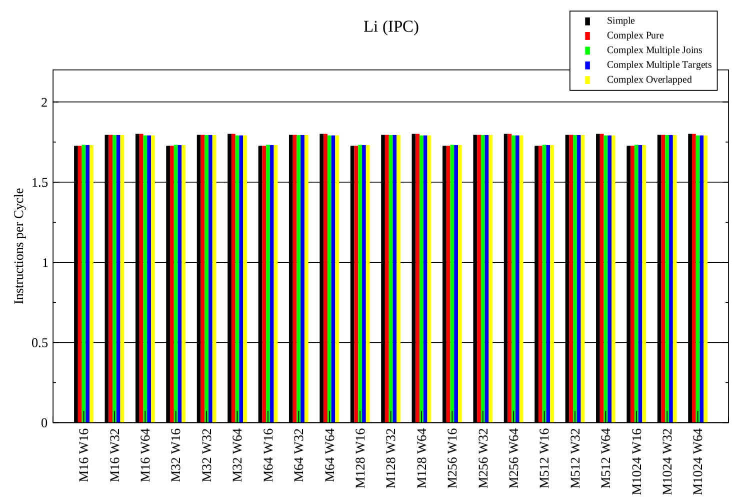


Figure 8.12: Li - IPC varying number of mapping tables

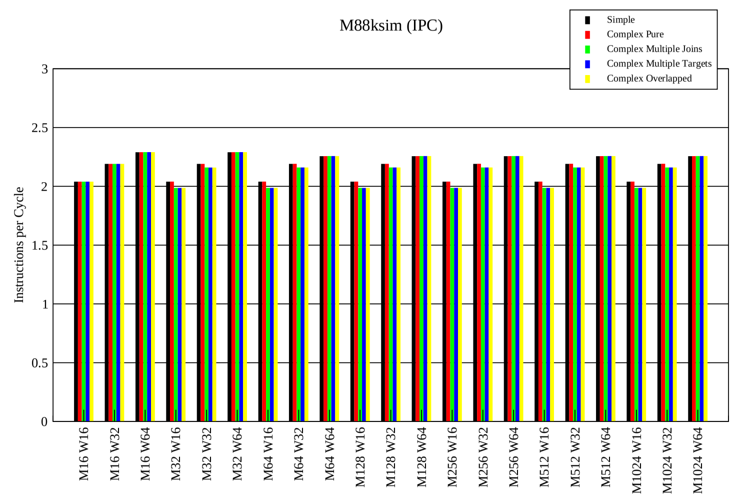


Figure 8.13: M88ksim - IPC varying number of mapping tables

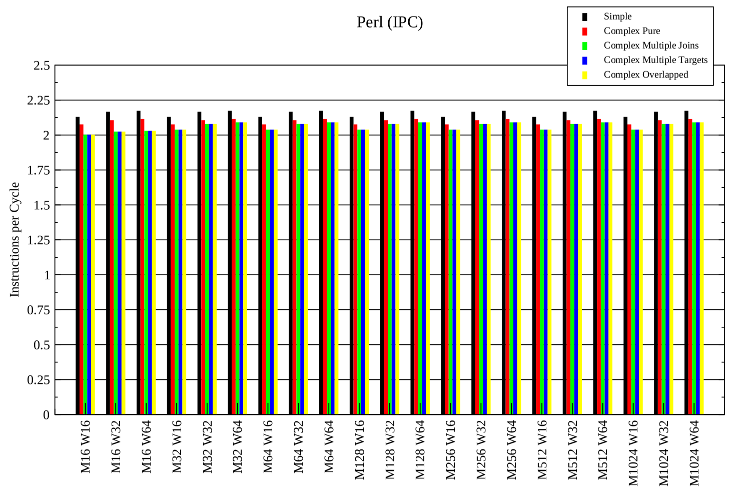


Figure 8.14: Perl - IPC varying number of mapping tables

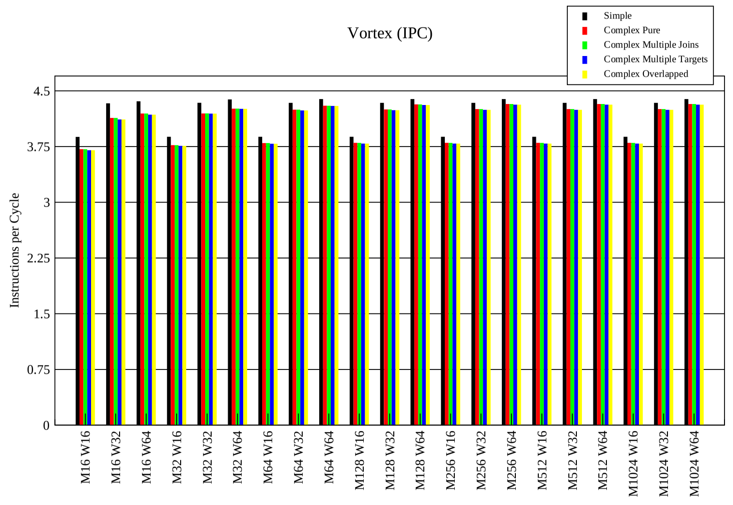


Figure 8.15: Vortex - IPC varying number of mapping tables